UNIVERSIDAD MICHOACANA DE SAN NICOLÁS DE HIDALGO

FACULTAD DE INGENIERÍA ELÉCTRICA

POSGRADO

HERRAMIENTA PARA EL ANÁLISIS DE SISTEMAS DINÁMICOS MEDIANTE DIAGRAMAS DE BIFURCACIÓN BASADO EN METAHEURÍSTICAS

TESIS

QUE PARA OBTENER EL GRADO DE
MAESTRO EN CIENCIAS DE LA INGENIERÍA ELÉCTRICA

PRESENTA
MAURICIO LÓPEZ CUEVAS VILLANUEVA

DIRECTOR DE TESIS
DR. JUAN JOSÉ FLORES ROMERO

CO-DIRECTOR DE TESIS
DR. CLAUDIO FUERTE ESQUIVEL

MORELIA, MICH. SEPTIEMBRE DE 2010

HERRAMIENTA PARA EL ANÁLISIS DE SISTEMAS DINÁMICOS MEDIANTE DIAGRAMAS DE BIFURCACIÓN BASADO EN METAHURÍSTICAS.

TESIS

Que para obtener el grado de

MAESTRO EN CIENCIAS EN INGENIERÍA ELÉCTRICA

presenta

Mauricio López Cuevas Villanueva

Dr. Juan José Flores Romero

Director de Tesis

Dr. Claudio Fuerte Esquivel

Co-Director de Tesis

Universidad Michoacana de San Nicolás de Hidalgo

Agosto 2010

Resumen

El ánalisis de sistemas dinámicos es de gran interés en muchas áreas de la ciencia. El trazo de diagramas de bifurcación es muy importante en este tipo de análisis. Los diagramas de bifurcación ayudan a predecir cualitativamente comportamientos dinámicos complejos en la estructura de un sistema, cuando existe variación en sus parámetros.

Los métodos tradicionales para el trazo de diagramas de bifurcación requieren de valores de inicio y ajuste de parámetros para su correcto funcionamiento. Los valores de estos parámetros regularmente no se conocen y requieren de un conocimiento detallado del sistema o de una búsqueda no sistemática de valores para éstos.

Como alternativa a los métodos tradicionales están los métodos metaheurísticos utilizados como complemento en el trazo de diagramas de bifurcación. Las técnicas metaheurísticas presentan una serie de ventajas y desventajas sobre los métodos tradicionales.

La intención de la presente tesis de maestría es la implementación de una herramienta auxiliar en el trazo de diagramas de bifurcación a través del uso de técnicas metaheurísticas utilizadas para la búsqueda de puntos fijos. Se diseñó e implementó un sistema que provee de una interfaz intuitiva y del conjunto de herramientas que permitan ingresar un sistema de ecuaciones diferenciales, generar diagramas de bifurcación y administrar de forma sencilla los archivos generados.

La técnica incorporada por defecto en el sistema desarrollado es la de optimización de enjambre de partículas con nichos, propuesta por Julio Agustín Barrera Mendoza en su tesis doctoral. Sin embargo, se desarrolló una estructura que permite, la adición de nuevos módulos, de tal forma que el problema de la localización de puntos fijos de un sistema puede ser resuelto por medio de distintas técnicas metaheurísticas.

Una vez calculados los puntos fijos del sistema, el sistema traza un diagrama de bifurcación, proporcionando una serie de herramientas (zoom, selección de áreas del gráfico, etc.) que permiten manipular el gráfico para su mejor análisis.

Abstract

Many research areas are interested in bifurcation analysis of dynamical systems. Bifurcation diagrams plotting is a very important task in this kind of analysis because it helps to qualitatively predict complex behaviors in the structure of a system where there is variation in its parameters.

Common bifurcation diagrams plotting methods, require of initial values and parameter adjustment for its correct operation. These values are frequently unknown, and requiere a deep knowledge of the system being analyzed or a non-systematic search of the values for these parameters.

As an alternative to common methods, metaheuristic methods are used as a complementary tool in bifurcation diagrams plotting. This kind of methods present a number of advantages and disadvantages over traditional methods.

The intention of this thesis is the implementation of an useful original software for bifurcation diagrams through the use of metaheuristic techniques in the search for equilibrium points of a dynamical system. A software design that provides to end users an intuitive inteface and a set of tools for entry systems of ordinary differential equations, generate bifurcation diagrams according to parameters set, and allow a simple manipulation of all these elements.

The technique included by default is particle swarm optimization, proposed by Julio Agustín Barrera Mendoza in his doctoral thesis. However, the program structure allows the addition of new modules where each of them can implement a different metaheuristic technique for fixed points calculation.

Once the fixed points of a system are calculated, the program can generate bifurcation diagrams in different ranges which can be manipulated by the user in different ways.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Algoritmos

Capítulo 1

Introducción

El proceso de la construcción de un diagrama de bifurcación está comienza con el proceso de búsqueda de puntos fijos del sistema. Un punto fijo a nivel físico es el estado en el que el sistema se encuentra en equilibrio, de ahí que los términos punto fijo y punto de equilibrio sean utilizados de manera indistinta. En general, la búsqueda de puntos fijos se realiza mediante herramientas de análisis numérico y algúna variante del método de continuación, que requiere que al menos una de las soluciones sea conocida; es decir, el primer punto de equilibrio. A partir de este punto se calcula el resto de la curva, mediante el método de Newton o variaciones de éste. Lo anterior implica que las funciones examinadas sean continuas y que debe existir un conocimiento previo de algún punto de equilibrio del sistema, lo cual regularmente no ocurre. Además de que existen casos en los que no se presenta la convergencia a una solución y es necesario el ajuste de varios parámetros para que los métodos funcionen de manera adecuada. Algunos de estos parámetros son sensibles y no ajustarlos adecuadamente propicia la no convergencia a un punto o acarrea un error.

Debido a que el trabajo inicial se reduce a la búsqueda de puntos de equilibrio que dicho de otra manera es encontrar las raíces del sistema de ecuaciones algebraicas del sistema dinámico, es posible plantear este problema como un problema de busqueda de óptimos y utilizar algoritmos que implementen alguna técnica metaheurística. Esta idea no es nueva y existen trabajos diversos en esta rama, como el de Elisha Sacks, en el cual se logra determinar el diagrama de fase de un sistema de dos ecuaciones lineales con un parámetro [Sacks91].

Dentro del posgrado de ingeniería eléctrica de la Universidad Michoacana de San Nicolás de Hidalgo, también existen trabajos al respecto, como el de Julio Barrera [Barrera08] que precisamente hace un análisis de la utilización de técnicas metaheurísticas aplicadas al análisis de sistemas dinámicos o como el trabajo de Rodrigo López [López10] que plantea la utilización de metaheurísticas en el análisis de sistemas dinámicos con discontinuidades y elementos discretos.

El grado de aplicación de este tipo de trabajos, se ve un tanto supeditado al conocimiento del lenguaje en que fue desarrollado o a su integración con otra herramienta para poder análizar los resultados. Así surgió la idea del desarrollo de una plataforma que permita dar a conocer estos y otros trabajos relacionados, integrando la introducción de modelos, el cálculo de sus puntos fijos, el trazo de su diagrama de bifurcación y herramientas para su almacenamiento y administración, con el propósito de ampliar el nicho de aplicación de este tipo de técnicas al análisis de sistemas dinámicos. Los beneficios del sistema pueden verse desde dos perspectivas. Primero, posibilitando a investigadores y programadores integrar a la plataforma sus distintas técnicas metaheurísticas y algoritmos de búsqueda de puntos de equilibrio sin necesidad de implementar los mecanismos tanto de introducción de modelos como del trazo del diagrama de bifurcación. Y segundo, brindando a los usuarios finales las ventajas (y desventajas) de la utilización de este tipo de técnicas, como el poco conocimiento previo del comportamiento del sistema que se requiere (entre otras), en el trazo de diagramas de bifurcación.

1.1. Estado del arte

En la actualidad, AUTO [Doedel09] es el software más utilizado para el trazo de diagramas de bifurcación. AUTO es una herramienta muy completa desarrollada en FORTRAN que utiliza el método de continuación para encontrar los puntos de equilibrio de un sistema de ecuaciones diferenciales ordinarias. Este método tiene la desventaja de que para poder encontrar el j-ésimo punto de equilibrio es necesario conocer el (j-1)-ésimo punto de equilibrio. Es decir, se requiere conocer el valor del punto de equilibrio a partir

del cual se desea comenzar a trazar el diagrama de bifurcación. Ésto impide encontrar los puntos de equilibrio de un sistema en una región determinada. La descripción de los métodos numéricos para el trazo de diagramas de bifurcación puede ser encontrada en los textos de Willy Govaerts [Govaerst00] y Yuri Kuznetsov [Kuznetsov98].

El sistema *sawThoot* [Regis99] permite el análisis de sistemas dinámicos discretos mediante diagramas de bifurcación, pero sólo permite el análisis de sistemas definidos por funciones de la circunferencia, conocidas como familia *Arnol*.

El algoritmo utilizado por el sistema AUTO utiliza una variante del método de Newton para encontrar las raíces del sistema y no puede trazar los diagramas de bifurcación de sistemas cuyas funciones no son diferenciables. Sin embargo, es un sistema muy completo, el cual además de trazar los diagramas de bifurcación también proporciona información acerca tanto de la estabilidad del punto fijo como del tipo de bifurcación observada.

El uso de técnicas metaheurísticas utilizadas en el análisis de sistemas dinámicos ha sido tratado anteriormente, v.g. el trabajo de Elisha Sacks, en el cual se logra determinar el diagrama de fase de un sistema de dos ecuaciones lineales con un parámetro [Sacks91]. O el trabajo de Varum Aggarwal [Aggarwal00] y Crina Grosan [Crina Grosan06], que presentan propuestas para la búsqueda de soluciones de sistemas de ecuaciones no lineales.

En el campo de las técnicas metaheurísticas, existen en la actualidad gran variedad de ellas. Los algoritmos genéticos han sido usados en una gran número de problemas de optimización [Haupt04]; la característica principal de los algoritmos genéticos es su convergencia a un óptimo global; incluso en problemas donde los algoritmos tradicionales no convergen. Esta característica no es deseable cuando necesitamos encontrar más de un óptimo; éste es el caso de los problemas multimodales.

Para resolver este tipo de problemas, existe un tipo de algoritmo metaheurístico que está basado en la observación de que en la naturaleza muchas especies coexisten en una región geográfica cerrada, compartiendo recursos y creando nichos, esos algoritmos genéticos

son llamados algoritmos de nichos [Mahfoud95]. Entre este tipo de algoritmos se encuentra la técnica llamada optimización de en enjambre de partículas (*PSO* por sus siglas en inglés).

Para la resolución de problemas multiobjetivo existe un algoritmo llamado *Evolución Diferencial*, desarrollado por Storn y Price [Storn Rainer95], para optimización en espacios continuos. En el algoritmo propuesto, se seleccionan tres individuos como padres. Uno de los padres es el principal y éste se perturba con el vector de diferencia de los otros dos padres. De esta manera se van conformando las nuevas soluciones.

Los sistemas inmunes artificiales [J. Timmis08] son una metaheurística inspirada por la inmunología teórica y las funciones inmunes observadas que se aplican a dominios de problemas complejos. Se han utilizado en tareas de recononocimiento de patrones y de optimización.

En la actualidad existe una gran diversidad de técnicas metaheurísticas y variantes de éstas que pueden ser aplicadas en el problema de búsqueda de puntos fijos de un sistema dinámico planteado como un problema de optimización multimodal.

1.2. Justificación

La implmentación de este sistema beneficia a investigadores en el área de optimización inteligente, debido a que se proporciona un marco de trabajo en el que el investigador sólo debe preocuparse por el desarrollo de una metaheurística de optimización. Una plataforma como la desarrollada, permite dar a conocer a mayor número de personas, los trabajos realizados en el área de optimización inteligente aplicadada al análisis de sistemas dinámicos, ya que en ocasiones, estos trabajos de investigación son desarrollados en lenguajes cuyo conocimento no es muy extendido (y su utilizacón requiere de conocimiento del lenguaje por parte del usuario) y en otras ocasiones es necesario que el usuario se apoye de otras herramientas (como de graficación por ejemplo) para complementar el uso de estos trabajos.

Los usuarios del sistema desarrollado contarán con las ventajas de utilizar técnicas metaheurísticas para la búsqueda de puntos de equilibrio, como poco conocimento previo del comportamiento de un sistema dinámico, la generación de diagramas de manera no supervisada, la posibilidad de analizar funciones que no son diferenciables o con discontinuidades.

El diseño de la aplicación incorpora diversas herramientas de gran potencial y a su vez poco difundidas (ANTLR, API Java Reflection, BeanShell y JMathPlot). Por lo que el trabajo puede ser útil a manera de tutorial en el uso de alguna de estas herramientas.

1.3. Hipótesis

Plantear el problema de la búsqueda de puntos de equilibrio de un sistema dinámico como un problema de búsqueda de máximos en una función que mapea la aptitud de los posibles puntos fijos del sistema en un rango bien definido convirtiéndolo en un problema de optimización, permite utilizar técnicas alternativas a los métodos de optimización tradicionales. Las llamadas técnicas metaheurísticas proveen una serie de ventajas y desventajas con respecto a los métodos tradicionales. La precisión (y calidad) de los resultados obtenidos por la técnicas metaheurísticas pueden medirse a través de una serie de factores: el número de óptimos globales que tiende a encontrar la técnica, propensión a estancarse en óptimos locales, duplicidad de óptimos, etc. El impacto que estos factores tienen en la generación de los diagramas de bifurcación es alto, por lo que distintas técnicas utilizadas generarán diagramas con distinta precisión.

La implementación de una herramienta que permita generar diagramas de bifurcación a través del uso de distintas técnicas metaheurísticas, proporcionará un marco que auxiliaría en la medición del desempeño de este tipo de técnicas.

Algunas de las ventajas de las técnicas metaheuríticas utilizadas como una herramienta auxiliar en el análisis de sistemas dinámicos son: la reducción del conocimento previo requerido del comportamiento del sistema que se analiza; reduce la supervisión re-

querida por parte del investigador en la generación de los diagramas de bifurcación. Por lo que una herramienta que integre la captura y almacenamiento de sistemas dinámicos además de la generación de diagramas de bifurcación, es capaz de reducir los tiempos de estudio de sistemas complejos.

1.4. Objetivos

La presente tesis tiene como propósito la implementación de un sistema para el trazo de diagramas de bifurcación que utiliza técnicas metaheurísticas en la búsqueda de puntos de equilibrio de sistemas de ecuaciones diferenciales ordinarias.

El sistema desarrollado debe contener las siguientes características mínimas:

- Permitir la definición de sistemas dinámicos.
- Evaluar un sistema dinámico con un vector de estado dado.
- Generar diagramas de bifurcación de dos y tres dimensiones utilizando técnicas metaheurísticas (PSO por defecto).
- Herramientas para la manipulación de las características visuales de los diagramas de bifurcación.
- Almacenamiento de diagramas y modelos.
- Interfaz para la navegación a través de diagramas y modelos

El objetivo de la implementación, fue el de proporcionar una herramienta útil (desarrollada en Java) para el análisis de sistemas dinámicos mediante el trazo de diagramas de bifurcación. El sistema esta constituido por: 1) Un compilador que permite la introducción (por parte del usuario) y el reconocimiento (por parte del sistema) de los sistemas dinámicos que se desean estudiar. El compilador traduce los modelos de un lenguaje de definición de sistemas dinámicos a lenguaje Java. 2) Un mecanismo que permite la ejecución de código Java de forma dinámica, es decir, ejecutar codigo Java que no se encuentra

en ningun archivo fuente, con el fin de evaluar los sistemas dinámicos. 3) Un mecanismo de programación reflexiva que permite a usuarios avanzados (programadores) desarrollar metaheurísticas e incorporarlas al sistema para que puedan ser utilizadas por los usuarios. 4) Una metaheurística usada por defecto por la aplicación para el cálculo de puntos fijos del sistema; el algoritmo de optimización de enjambre de partículas propuesto en la tesis doctoral de Julio Barrera [Barrera08]. 5) Un mecanismo de generación y manipulación de diagramas de bifurcación de dos y tres dimensiones. 6) Una interfaz gráfica que permite la administración y almacenamiento de los modelos y diagramas del usuario.

A través de los diversos capítulos se definen los conceptos teóricos y lineamientos técnicos con los que se construyó cada uno de los módulos de la aplicación, de tal manera que esta tesis pueda ser utilizada como una guía de referencia tanto de uso de la aplicación, como un manual técnico útil para modificaciones y posteriores versiones de la aplicación.

1.5. Contenido de la tesis

En el Capítulo 2 se enuncia el problema del análisis de sistemas dinámicos mediante el trazo de diagramas de bifurcación.

En el Capítulo 3 se decribe el compilador y el mecanismo de introducción y ejecución de modelos, así como la implementación del análisis léxico, sintáctico y semántico de los mismos.

El Capítulo 4 describe el algoritmo de optimización de enjambre de párticulas implementado como la técnica utilizada por defecto por la aplicación para el cálculo de puntos de equilibrio. Se describe también el mecanismo desarrollado para la adición de nuevos módulos de cálculo de puntos de equilibrio.

El Capítulo 5 describe el mecanismo del trazo de los diagramas de bifurcación y las herramientas utilizadas en su desarrollo.

El Capítulo 6 muestra los resultado obtenidos mediante el uso de la aplicación.

Finalmente, en el Capítulo 7 se presentan las conclusiones, así como el desarrollo futuro a partir de la presente tesis.

Capítulo 2

Sistemas dinámicos

Se podría decir que los sistemas dinámicos son un área "joven"de las matemáticas, aunque se remontan a Newton y sus estudios de mecánica Celeste y a Henri Poincaré, quien inició el estudio cualitativo de las ecuaciones diferenciales. Sin embargo, fue hace unos 40 años que los sistemas dinámicos se establecieron como un área propiamente dicha, gracias al trabajo de matemáticos e ingenieros como Smale, Arnold, Lyapunov, etc [Kuznetsov98].

Este capítulo presenta un breve panorama de los conceptos de sistemas dinámicos. Se describe el método tradicional para el trazo de diagramas de bifurcación y sus implicaciones.

2.1. Definiciones

Si tratamos de precisar el concepto de sistema dinámico, podríamos decir de una manera muy general, que se trata de un sistema determinista. Es decir, consideramos situaciones que dependan de algún parámetro dado, que frecuentemente se asume es el tiempo (sin embargo esta consideración no es correcta) y que varían de acuerdo a leyes establecidas que gobiernan su comportamiento. De manera que el conocimiento de su estado en un momento dado nos permite reconstruir el pasado y predecir el futuro de dicho sistema. Si las leyes que gobiernan su evolución no cambian en el tiempo podemos considerar el comportamiento de tal sistema como completamente definido por su estado inicial. La noción

de un sistema dinámico incluye un conjunto de posibles estados y una ley de evolución de un estado en el tiempo. En general, un sistema dinámico es descrito como un sistema de ecuaciones diferenciales de primer orden.

Bifurcación es la acción de separarse en varias partes y se utiliza en varios contextos para definir la situación en la que la topología y el comportamiento del objeto bajo estudio se altera con el cambio del valor de alguno de los parámetros de los cuales depende el objeto. En ese sentido un diagrama de bifurcación es la representación gráfica del comportamiento de alguno de los componentes del sistema en relación a la variación de uno o más de sus parámetros. Un diagrama de bifurcación suele mostrar el número de puntos de equilibrio y la posición de estos respecto a alguna variable del sistema en relación al cambio en alguno de los parámetros del sistema [Seidel99].

Uno de los métodos de análisis de sistemas dinámicos consiste en determinar los puntos de bifurcación. Esto es realizado regularmente en dos partes: la primera consiste en determinar un punto fijo estable del sistema. Los puntos fijos estables se encuentran regularmente primero determinando los puntos fijos de un sistema y después determinando la estabilidad del punto fijo encontrado. Este punto estable es después utilizado como punto inicial para el método llamado *de continuación*, el cual, como su nombre lo indica, consiste en construir a partir del punto inicial un diagrama donde se sigue la evolución del punto fijo conforme uno o más de los parámetros del sistema son modificados.

Si el sistema es grande, relativamente complejo o no se tiene mucho conocimiento acerca de éste, la búsqueda del primer punto de equilibrio del sistema se convierte en una tarea realizada de una manera no sistemática y no automatizada. Para continuar con el trazo del diagrama es necesario que las funciones que componen el sistema sean continuas y estén definidas a lo largo del rango en donde se desea analizar el sistema.

2.2. Puntos fijos

De manera general, un punto fijo (también conocido como punto de equilibrio) de un sistema dinámico, es el punto correspondiente al estado del sistema que permanece constante en el tiempo; es decir, en donde el estado no presenta ningun cambio. Anteriormente se comentó que un sistema dinámico es descrito generalmente como un sistema de ecuaciones diferenciales, entonces un punto fijo es el conjunto de valores (un valor asociado a cada una de las variables del sistema) que cumple con la condición de que todas las diferenciales del sistema son iguales a cero [Seidel99]. Esto es, si un sistema dinámico es descrito por el sistema de ecuaciones mostrado en la Ecuacion 2.1

$$\begin{aligned} \dot{x_1} &= f_1(x_1, x_2, \cdots, x_n) \\ \dot{x_2} &= f_2(x_1, x_2, \cdots, x_n) \\ &\vdots \\ \dot{x_n} &= f_n(x_1, x_2, \cdots, x_n) \end{aligned} \tag{2.1}$$

necesitamos encontrar el un vector $(x_1^*, x_2^*, \ldots, x_n^*)$ tal que

$$\begin{aligned} 0 &= f_1(x_1^*, x_2^*, \cdots, x_n^*) \\ 0 &= f_2(x_1^*, x_2^*, \cdots, x_n^*) \\ &\vdots \\ 0 &= f_n(x_1^*, x_2^*, \cdots, x_n^*) \end{aligned} \tag{2.2}$$

Este vector no necesariamente existe para valores reales de las variables x_i y no necesariamente es único en la región de operación del sistema dinámico. El sistema de ecuaciones 2.2 es llamado también sistema de ecuaciones algebraicas del sistema.

El sistema presentado en la Ecuacion 2.1 no contiene parámetros. Sin embargo, de manera general, los sistemas dinámicos también dependen de parámetros que modifican

su comportamiento, los cuales son expresados como se muestra en la Ecuación 2.3.

$$\begin{aligned} \dot{x_1} &= f_1(x_1, x_2, \cdots, x_n, \alpha, \beta, \cdots) \\ \dot{x_2} &= f_2(x_1, x_2, \cdots, x_n, \alpha, \beta, \cdots) \\ &\vdots \\ \dot{x_n} &= f_n(x_1, x_2, \cdots, x_n, \alpha, \beta, \cdots) \end{aligned} \tag{2.3}$$

donde $\alpha, \beta, \cdots$ representan los parámetros del sistema. De manera general se buscan puntos fijos del sistema representado por la Ecuación 2.3 donde se tienen uno o más parámetros que varían en rangos determinados.

Un punto de equilibrio se considera estable si todas las perturbaciones suficientemente pequeñas que se realicen al sistema terminan por amortiguarse con el tiempo. Es decir, si tomamos un punto cercano al punto de equilibrio estable y seguimos su trayectoria respecto al tiempo, éste eventualmente convergerá al punto fijo. De manera inversa, un punto fijo se considera inestable si una perturbación que se realiza en el punto tiende a aumentar con el tiempo. Al seguir la trayectoria con el tiempo de un punto cercano a un punto inestable, éste se alejará del punto fijo [Kuznetsov98].

En el sentido de Lyapunov la estabilidad se caracteriza de la siguiente manera. Considere el sistema

$$\dot{x} = f(x) \tag{2.4}$$

donde $f : D \to R^n$ es un mapa localmente Lipschitz desde un dominio $D \subset R^n$ en R^n. Supongamos que $\bar{x} \in D$ es un punto fijo del Sistema 2.4, es decir $f(\bar{x}) = 0$. Consideremos que $\bar{x} = 0$. Entonces el punto fijo $x = 0$ del sistema 2.4 es:

- *estable*, si para cada $\epsilon > 0$ existe $\delta = \delta(\epsilon)$ tal que

$$||x(0)|| < \delta \Rightarrow ||x(t)|| < \epsilon, \forall t \geq 0$$

- *inestable* si no es estable.

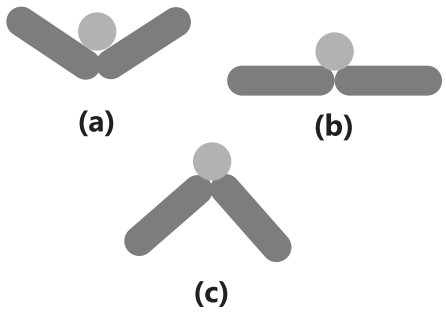


Figura 2.1: Varios estados de una pelota en un puente de compuertas con varias condiciones de inclinación: a)Punto fijo estable, b)Aparición de puntos fijos estables c) Aparición de un solo punto fijo inestable.

- *asintóticamente estable* si es estable y δ puede elegirse tal que

$$||x(0)|| < \delta \Rightarrow \lim_{t\to\infty} x(t) = 0$$

2.3. Bifurcaciones

Una bifurcación es un cambio en la estructura cualitativa o topológica de un objeto determinado. Para nuestro caso de estudio, el objeto es el sistema dinámico; un cambio en la estructura cualitativa representa un cambio en la estabilidad de un punto fijo (puede pasar de estable a inestable o viceversa, además de que existen otros comportamientos: semiestable, asintóticamente estable, etc.) y un cambio en la estructura topológica, representa la aparición o desaparición de puntos fijos en el sistema. Para comprender mejor este concepto, consideremos el ejemplo de un objeto esférico (una pelota pordría ser) colocado en un puente de compuertas móviles (ver la Figura 2.1). Si las plataformas del puente se encuentran hacia abajo formando un ángulo menor a los 180 grados ($\theta < 180$)con respecto a la parte superior de la figura (Figura 2.1(a)) la pelota queda en la frontera de las dos platafomas. Ahí, si la pelota es desplazada por cualquier fuerza (cierta distancia) eventual-

mente siempre regresará a la posición inicial (entre las dos compuertas). De esta manera llamaríamos a esa posición de la pelota un punto fijo estable. Hay que observar que si las plataformas comienzan a subir, mientras el ángulo formado por las plataformas con respecto a la parte superior sea menor a 180 grados ($\theta < 180$), la posición de la pelota seguirá estando en un punto fijo estable. Sin embargo, si el ángulo formado por las plataformas llega a 180 grados ($\theta = 180$), lo que sucede es que toda la superficie de la plataforma se convierte en un conjunto de puntos fijos estáticos (es decir, que en cualquier posición que coloquemos la pelota, está permanecera ahí) (ver la Figura 2.1(b)). Una vez que la plataforma forme un ángulo mayor a 180 grados ($\theta > 180$) con respecto a su parte superior. (ver Figura 2.1(c)), el punto que se encuentra entre las dos plataformas se convirete en un punto fijo inestable. Si la pelota es colocada en este punto, permanecerá en ese lugar, sin embargo cualquier perturbación ocasionará que la pelota ruede hacia fuera de la plataforma.

Supongamos que la posición de la pelota es modelada por alguna función f. Podemos decir, de manera poco formal, que existe una solución de la función f para los valores del ángulo $\theta < 180$ y que esta solución será atraída hacia el punto central entre las dos plataformas, y que esta solución cambia para $\theta = 180$ y vuelve a cambiar para $\theta > 180$. Observamos que θ es lo que es llamado parámetro del sistema y la posición de la pelota es lo que es conocido como el estado del sistema. En un problema práctico es necesario encontrar los valores de los parámetros de un sistema para los cuales ocurren transiciones tanto cuantitativas como cualitativas (Bifurcaciones).

De manera formal, considere un sistema dinámico que depende de un parámetro α. En el caso de un sistema continuo no autónomo, los escribiremos como se muestra en la Ecuación 2.5

$$\dot{x} = f(x, \alpha), \tag{2.5}$$

donde $x \in R^n$ y $\alpha \in R^m$ representan las variables de estado y parámetros, respectivamente. Considere el diagrama de fase del sistema que representa la Ecuación (2.5); al variar un parámetro, el diagrama de fase también cambia. Hay tres posibilidades: el sistema puede permanecer topológicamente equivalente al sistema original, ocurre un cambio cualitativo;

o su topología cambia, ocurre un cambio topológico.

La aparición de un diagrama de fase no equivalente topológicamente bajo la variación de un parámetro es llamada bifurcación. Así, una bifurcación es un cambio en el tipo de topología del sistema, cuando éste pasa a través de un valor crítico (valor de bifurcación).

2.4. Método del trazo del diagrama de bifurcación

Como ya se mencionó, los métodos de continuación han sido utilizados para el trazo de diagramas de bifurcación. Los diagramas generados por este tipo de métodos son curvas unidimensionles definidas por n ecuaciones en espacios de dimensión $n+1$. En particular se han utilizado para identificar la curva de equilibrios definida por el sistema de n ecuaciones

$$f(x, \alpha) = 0, x \in R^n, \alpha \in R \tag{2.6}$$

definidas en un sistema dinámico continuo en R^{n+1}.

La mayoría de los métodos de continuación usados en el análisis de bifurcación implementan métodos de predicción-corrección. Un estudio detallado de estos métodos puede consultarse en [Seidel99].

La Ecuación (2.6) define curvas con soluciones que se supone que existen. También se asume que al menos una de esas soluciones es conocida; es decir, se conoce un punto de equilibrio (x_1, α_1). El problema de continuación consiste en calcular el resto de los puntos de equilibrio hasta localizar un punto deseado.

$$(x_2, \alpha_2), (x_3, \alpha_3), \ldots$$

El j-ésimo paso del proceso de continuación se inicia desde una solución (x_j, α_j) de la Ecuación (2.6) para un valor del parámetro α_j y el objetivo es calcular la solución (x_{j+1}, α_{j+1}) para el siguiente valor del parámetro α_{j+1}.

El paso de j a $j+1$ se divide en dos:

$$(x_j, \alpha_j) \xrightarrow{predictor} (\bar{x}_{j+1}, \bar{\alpha}_{j+1}) \xrightarrow{corrector} (x_{j+1}, \alpha_{j+1})$$

tal como se muestra en la Figura 2.2. En general la predicción $(\bar{x}, \bar{\alpha})$ no es una solución

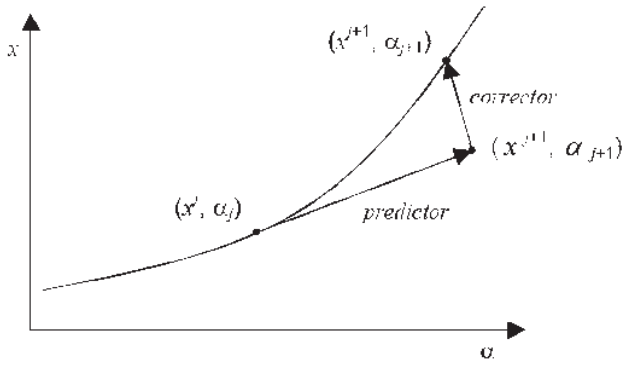


Figura 2.2: Método de continuación *Predictor - Corrector*

de la Ecuación 2.6. El predictor simplemente identifica un punto que el corrector toma como inicio de sus iteraciones para encontrar finalmente la solución. La distancia entre dos soluciones consecutivas (x_j, α_j) y (x_{j+1}, α_{j+1}) se llama tamaño de paso.

La mayoría de los algoritmos que implementan métodos de continuación basados en predictor - corrector para detección de bifurcaciones, incluyen tres fases fundamentales que ejecutan en cada iteración:

1. Predicción
2. Corrección
3. Control de paso

Los métodos de predicción pueden ser divididos en dos grupos: métodos ODE y métodos de extrapolación polinomial. Los primeros se basan en la función $f(x, \alpha)$ y sus derivadas, mientras los segundos sólo utilizan las soluciones (x, α) de la Ecuacion (2.6). El desempeño óptimo del algoritmo de continuación depende en gran medida del tamaño de

paso. El control de paso por lo general está basado en estimaciones sobre la convergencia del corrector.

El proceso de continuación puede hacerse lento si se selecciona un tamaño de paso demasiado corto (hay que evaluar demasiados puntos sobre la curva), o bien si el tamaño de paso es demasiado largo (puede que el corrector no converja o se demore en encontrar la solución).

2.5. Comentarios finales

No existen al momento programas de análisis de sistemas dinámicos que permitan el trazo de un diagrama de bifurcación variando dos parámetros al mismo tiempo, además de que es necesario que el punto inicial para el trazo del diagrama sea un punto estable del sistema. El trabajo propone una herramienta alternativa para el trazo de diagramas de bifurcación sin necesidad de iniciar el trazo desde un punto estable, lo único que necesita conocer el usuario es el rango de operación del sistema. Además permite la variación de más de un parámetro en el mismo análisis y herramientas para la manipulación del diagrama de bifurcación, todo esto utilizando técnicas de optimización dinámica (metaheurísticas).

El sistema desarrollado genera diagramas de bifurcación para sistemas de ecuaciones diferenciales ordinarias de primer orden (sistemas dinámicos continuos). Los sistemas de ecuaciones pueden contener variables algebraicas y cualquier número de parámetros. La grámatica usada para definir este tipo de sistemas se describe en el siguiente capítulo.

El tema de la estabilidad no es tratado con más detalle en esta tesis, debido a que la primera versión del sistema no implementa el cálculo de la estabilidad de los puntos fijos. Y se plantea en la sección de trabajo futuro.

Capítulo 3

Compilador de sistemas de ecuaciones diferenciales

Este capítulo presenta algunos conceptos básico de teoría de compiladores y describe la implementación realizada de un compilador de sistemas de ecuaciones diferenciales utilizado en el sistema desarrollado. Podría considerarse un tutorial sobre la sintaxis del lenguaje diseñado. La estrategia que se seguirá para lograr este propósito será la de presentar algunos conceptos básicos de compilación, describir algunas de las características de ANTLR (An Other Tool for Language Recognition, herramienta utilizada para la construcción del compilador) y decribir cada una de las diferentes fases de su desarrollo.

3.1. Conceptos básicos

En el sentido más amplio, un compilador es un programa de software cuya función es la de traducir código fuente escrito en lenguaje de alto nivel, a un código equivalente en otro lenguaje de nivel inferior (típicamente lenguaje máquina). En nuestro caso, el código de alto nivel será un lenguaje diseñado para hacer lo más natural posible la descripción de los sistemas dinámicos por parte del usuario (sistemas de ecuaciones diferenciales), mientras que el lenguaje objeto que generará el compilador, será una serie de instrucciones equivalentes en código Java, las cuales serán interpretadas por la máquina virtual de Java que ejecuta el

sistema.

3.1.1. Intérpretes y compiladores

Los lenguajes de programación que requieren ser traducidos por un compilador a código máquina se denominan *lenguajes compilados*. En oposición a los lenguajes compilados encontramos los lenguajes interpretados. Los programas escritos en lenguajes interpretados no se traducen sino hasta el momento de ser ejecutados. Para que esta ejecución sea posible, es necesario que un software adicional, llamado *intérprete*, los lea y ejecute sus instrucciones una a una. Un programa escrito en un lenguaje interpretado se ejecutará, por tanto, más lentamente que su equivalente compilado. Esto ocurre principalmente porque el programa tendrá que estar compartiendo el hardware con el intérprete.

3.1.2. El proceso de compilación

Todo lenguaje de programación está formado por un conjunto de *símbolos básicos*, que se agrupan para formar los elementos de un *vocabulario*, de la misma manera que en cualquier lengua las letras se agrupan para formar las palabras. Los símbolos de los lenguajes de programación son los caracteres que forman el código y a sus *palabras* les llamaremos *tokens* [Cota03]. Las reglas de un lenguaje indican cómo pueden o no agruparse los diferentes *tokens*. A estas reglas se las llama *reglas sintácticas*. Es frecuente que el vocabulario se defina implícitamente al definir las reglas sintácticas de un lenguaje. Al conjunto de reglas sintácticas de un lenguaje se la llama gramática. El conjunto de las reglas sintácticas del castellano es la gramática castellana.

Por último, un lenguaje suele ir acompañado de ciertas reglas que complementan a las reglas sintácticas. En el lenguaje natural utilizamos la lógica y la memoria para saber si una frase tiene "sentido". A esta parte se le conoce como semántica.

3.1.3. Fases del proceso de compilación

Lo primero que debe hacer un compilador es comprobar que la información que se le suministra pertenece a su lenguaje (que no hay errores léxicos, sintácticos ni semánticos).

Si es así, el intérprete debe representar de alguna manera la información que se le suministró para poder trabajar con ella y finalmente traducir dicha información a código objeto.

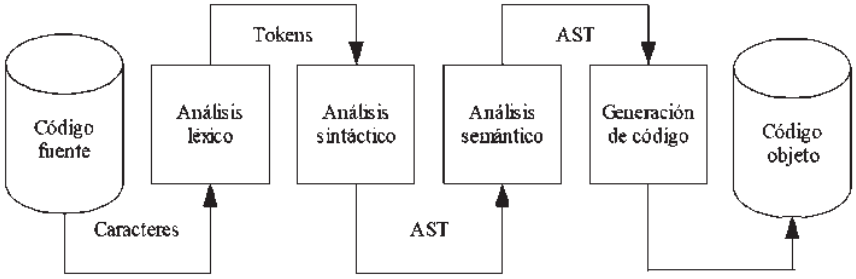


Figura 3.1: Fases del proceso de compilación

Un esquema de dicho funcionamiento es el que se muestra en la Figura 3.1. A primera vista podemos distinguir que hay dos tipos de elementos: los elementos activos (figuras cerradas) y los flujos de datos (representados como flechas), que unen los diferentes elementos activos. Si entre los elementos activos A y B existe una flecha llamada C, eso quiere decir que A produce el flujo de datos C, que es usado por B. A continuación analizaremos brevemente cada elemento y cada flujo del diagrama.

El código fuente suele componerse de uno o varios archivos de texto (también, pueden ser archivos binarios), en los que se almacena cierta información cuyo fin es provocar ciertas acciones en una máquina objetivo (que puede no ser la que está interpretándolos). Para ello, los archivos son leídos del disco y llevados a la memoria, conformando el flujo denominado *caracteres* (que también podría ser un flujo de bytes) [Parr93].

El análisis léxico por otra parte, es el que está relacionado con el *vocabulario* del que ya hablábamos anteriormente. El proceso del análisis léxico agrupa los diferentes caracteres de su flujo de entrada en *tokens*. Los *tokens* son los símbolos léxicos del lenguaje; se asemejan mucho a las palabras del lenguaje natural. Los *tokens* están identificados con símbolos (tienen un nombre) y suelen contener información adicional (como la cadena de

caracteres que los originó, el archivo en el que se encuentran, la línea donde comienzan, etc). Una vez que son identificados, son transmitidos al siguiente nivel de análisis. El programa que permite realizar el análisis léxico es un analizador léxico. En inglés se le suele llamar *scanner* o *lexer*.

En el análisis sintáctico se aplican las reglas sintácticas del lenguaje al flujo de *tokens*. En caso de no haberse detectado errores, el intérprete representará la información del código fuente en un *Árbol de Sintaxis Abstracta*, que no es más que una representación arbórea de los diferentes patrones sintácticos que se han encontrado al realizar el análisis. En este proceso se eliminan los elementos innecesarios (signos de puntuación, paréntesis, etc.). De aquí en adelante llamaremos AST a los *Árboles de Sintaxis Abstracta* (por sus siglas en inglés *Abstract Syntax Tree*) [Parr93].

El código que permite realizar el análisis sintáctico se llama *analizador sintáctico*. En inglés se le llama *parser*, que significa *iterador* o directamente *analyzer* (*analizador*).

El análisis semántico es realizado sobre el AST y empieza por detectar incoherencias a nivel sintáctico en el AST. Si el AST supera esta fase, es común enriquecerlo para realizar un nuevo análisis semántico. Es decir, es común efectuar varios análisis semánticos, cada uno centrado en aspectos distintos. Durante estos análisis, el árbol es modificado. Cualquier herramienta que realice un análisis semántico es llamada "analizador semántico" en este texto. En la bibliografía inglesa suelen referirse a los analizadores semánticos como *tree parsers* [Parr93].

La generación de código es la última etapa en el proceso de compilación y utiliza el AST enriquecido, producto del proceso de análisis semántico, para generar código objeto.

3.2. ANTLR (Another Tool for Language Recognition)

En el reconocedor de sistemas dinámicos desarrollado, las tres primeras fases del análisis son cubiertas por una herramienta para desarrollo de compiladores llamada ANTLR

(ANother Tool for Language Recognition).

ANTLR [Cota03] es un software desarrollado en Java por varios programadores, aunque la idea inicial y las decisiones principales de diseño son de Terence Parr [Sommers06]. En su proyecto de fin de licenciatura, Terence presentaba una manera eficiente de implementar los analizadores LL (los cuales son explicados en la seccion 3.2.4). Los hallazgos presentados en su tesis fueron los que le llevaron a implementar PCCTS (Purdue Compiler Construction Tool Set)[Parr93], que puede considerarse como la primera versión de ANTLR.

ANTLR es capaz de actuar a tres niveles del proceso de compilación, a diferencia de otras herramientas de desarrollo de compiladores, ver la Figura 3.2. El uso de una sola

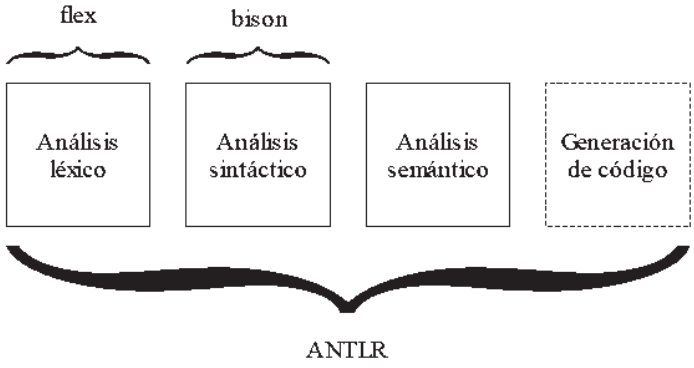


Figura 3.2: Comparación de ANTLR con otras herramientas

herramienta para todos los niveles de compilación tiene varias ventajas. La más importante es la estandarización. Con ANTLR basta con comprender el paradigma de análisis una vez para poder implementar todas las fases de análisis. Con *flex* y *bison* es necesario comprender y saber utilizar herramientas completamente diferentes. *Flex* está basado en autómatas finitos deterministas y *bison* es un analizador de izquierda a derecha, (Figura 3.2), además de que es necesario utilizar otras herramientas para realizar el análisis semántico.

3.2.1. Definición de reglas BNF (Backus-Naur Form)

Los lenguajes se especifican mediante un conjunto de reglas que es llamado gramática. ANTLR utiliza un lenguaje denominado EBNF (Extended Backus-Naur Form) que es una extensión de otro llamado BNF. En BNF todas las reglas comienzan por un *nombre* seguido por el caracter *dos puntos* (:), a continuación aparece el *cuerpo* de la regla. Y todas las reglas terminan con el caracter *punto y coma* (;).

```
nombre : cuerpo ;
```

Las reglas pueden ser de tres tipos: alternativas, enumeraciones y referencias a símbolos básicos del vocabulario o a otras reglas.

El tipo de regla más flexible es la alternativa. Las alternativas sirven para expresar elección entre varias opciones. Las diferentes opciones de la regla se separan con un caracter de barra vertical (|). Por ejemplo, para indicar que *un perro* puede ser un *foxterrier* o un *pastor*. Esta regla la podríamos escribir así:

```
perro: FOXTERRIER | PASTOR;
```

Los símbolos terminales y especialmente los *tokens* en el analizador sintáctico, se definen con letras mayúsculas, mientras que los nombres de las reglas se escriben con minúsculas. Además, los símbolos del vocabulario, se pueden utilizar en otras reglas. Por ejemplo:

```
cuadrupedo : perro
             | gato
             | caballo
             ;
perro : ... /* definir perro */ ;
gato : ... /* definir gato */ ;
caballo : ... /* definir caballo */ ;
```

Otro tipo de regla muy utilizado es la enumeración. Una enumeración no es más que una lista ordenada de referencias (a otras reglas o a elementos del vocabulario). Sirve para reconocer series de elementos. Los elementos simplemente se escriben unos detrás de otros, en el orden deseado y separados por espacios, retornos de carro o tabulaciones.

```
frase: EL cuadrupedo se_movia DEPRISA;
se_movia : CORRIA
         | GALOPABA
         ;
```

En este caso los símbolos básicos del lenguaje son EL, DEPRISA, CORRIA y GALOPABA (además de los nombres de perros citados anteriormente). La parte *frase* y *se_movia* son reglas de la gramática.

La regla *frase* permite reconocer muchas entradas. Por ejemplo reconocerá la entrada "EL FOXTERRIER CORRIA DEPRISA".

En BNF es posible utilizar enumeraciones como sub reglas de alternativas:

```
frase: EL perro PERSEGUIA AL gato
       | EL caballo COME PIENSO
       | UN cuadrupedo TIENE CUATRO PATAS
       ;
```

Los patrones repetitivos sirven para reconocer uno o más elementos o cero o más elementos. En BNF los patrones repetitivos deben implementarse con la recursión, es decir, con una regla que se llame a sí misma. Por ejemplo, para reconocer una llamada a una función con una lista de cero o más parámetros, se escribe:

```
llamada: NOMBRE PARENT_AB parametro PARENT_CE ;
parametro: argumento listaParametros
listaParametros: COMA argumento listaParametros // regla recursiva
| /* nada */
;
argumento : ENTERO | NOMBRE ;
```

Suponiendo que ENTERO represente cualquier número entero, NOMBRE cualquier identificador, PARENT_AB el paréntesis izquierdo ("("), PARENT_CE el paréntesis derecho (")") y COMA el símbolo coma (","), tendremos que la regla anterior permitirá reconocer entradas como $f(x)$, $max(a, 10)$ o $getMousePos()$. Nótese que para poder implementar este conjunto de reglas hemos tenido que utilizar una alternativa vacía.

3.2.2. EBNF (Extended Backus-Naur Form)

Como ya se mencionó, EBNF es una extensión de BNF(Extended Backus-Naur Form) [Scowen98] por lo tanto lo amplía en muchos aspectos, por ejemplo, en BNF no se permite el uso de alternativas como sub reglas de una enumeración. EBNF si lo permite. Para ello es recomendable utilizar paréntesis:

```
orden: PINTAR (RUEDAS|CHASIS) DE (AZUL|AMARILLO|VERDE);
```

Las subreglas admiten cualquier nivel de anidamiento en EBNF (puede insertarse una subregla dentro de una regla hasta donde se desee). Además de la capacidad de generar subreglas, EBNF supera ampliamente a BNF en el reconocimiento de patrones repetitivos. Para ello introduce dos operadores: *la clausura positiva* (que se representa con el símbolo "+") y *la cerradura de Kleene* (que se representa con el asterisco, "∗"). Estos dos operadores se emplean en conjunción con los paréntesis para indicar repetición. *La clausura positiva* indica una o mútiples apariciones de un patrón, mientras que el cierre de Kleene indica cero o múltiples apariciones de un patrón. Para el ejemplo anterior de la llamada a función podemos escribir

```
llamada: NOMBRE PARENT_AB (parametro)* PARENT_CE ;
parametro : ENTERO | NOMBRE ;
```

EBNF permite dos tipos adicionales de subreglas: la opcionalidad y la negación. Una sub regla opcional se representa con el operador de opcionalidad, que es el signo de interrogación "?". Normalmente se utiliza en conjunción con los paréntesis. La subregla opcional es una regla que puede estar o no en la entrada. Por ejemplo:

```
regla : UN perro (GRANDE) ? NO ES UN caballo ;
```

El *GRANDE* puede estar o no en la entrada que se analice (es opcional). El operador de opcionalidad no está limitado a símbolos básicos del lenguaje; se pueden insertar subreglas, referencias a otras reglas, etc.

El símbolo de la negación es la tilde "∼". Sirve para indicar que se espera una entrada que sea cualquiera que NO satisfaga esta regla. Normalmente se utiliza en los analizadores léxicos. Por ejemplo, una regla simple para reconocer comentarios en el analizador léxico sería:

```
comentario : "/*"
             (~("*/"))*
             "*/" ;
```

Un comentario comienza con la cadena "/*", seguido de cero o más veces (la cerradura de Kleene) cualquier cosa que no sea la cadena de terminación, "*/" y finalmente una cadena de terminación.

3.2.3. Acciones

La sintaxis EBNF sirve para definir la parte analítica de los reconocedores. Bastan las reglas EBNF para comprobar la adecuación de una entrada. No obstante, en la práctica, un reconocimiento se realiza con una intención práctica que va más allá de comprobar que la entrada se adapta a las reglas sintácticas. Por ejemplo, en muchas ocasiones deseamos traducir el código fuente a código máquina o a otro código (objeto). Para poder llevar a término la traducción, tanto BNF como EBNF introducen un nuevo tipo de elementos: las acciones.

Las acciones son bloques de código nativo que deben ejecutarse al llegar a ellas. Están separadas por llaves del resto del código:

```
r : A {System.out.println("Ha llegado una A");}
  | B {System.out.println("Ha llegado una B");}
  ;
```

3.2.4. Analizadores recursivos descendentes (LL)

Los algoritmos de análisis llamados *de arriba a abajo* son los algoritmos de análisis más intuitivos. Los analizadores recursivos descendentes son un conjunto de métodos mutuamente recursivos. En el momento en que el analizador requiere determinar cual de las

alternativas de una regla debe utilizar, el analizador se basa en el siguiente símbolo que se encuentra en la entrada. A dicho símbolo se le llama símbolo de *lookahead*. Si solamente necesita un símbolo para determinarlo, decimos que el lookahead es 1 y que el analizador es LL(1).

Los analizadores recursivos descendentes deben su nombre a que si se mira el análisis que realizan de forma arbórea, comienzan con la raíz de dicho árbol y van descendiendo por la estructura del árbol identificando estructuras cada vez más simples hasta llegar a los símbolos terminales, u hojas del árbol.

Los analizadores LL(1) deben poder predecir que patrón coincidirá con la entrada utilizando únicamente el primer token que podría ser utilizado en cada alternativa de la regla. Al conjunto de los primeros tokens de cada alternativa de una regla se le llama conjunto PRIMERO (regla).

Cuando los conjuntos PRIMERO de las diferentes alternativas de una regla no son disjuntos, se dice que hay una ambigüedad, o que la gramática no es determinista. Es decir, que al repetirse uno de los tokens al principio de más de una alternativa, no es posible conocer cual de las dos o más alternativas es la correcta utilizando un sólo símbolo de lookahead.

El concepto de LL(1) puede extenderse a LL(k), con k>1. Así, un analizador LL(2) no tendría dificultades con la gramática anterior. k es la letra por antonomasia para hablar del lookahead.

3.2.5. Analizador ANTLR

ANTLR usa una versión ampliada de LL(k) llamada pred-LL(K). Un analizador con lookahead variable, que normalmente trabaja con un valor de k pequeño (por defecto 1), ampliándolo solamente en las reglas que lo requieran. Para ello añade predicados especiales a las reglas que necesitan un lookahead mayor. Lo cual aumenta la potencia del algoritmo,

pudiéndose leer algunas gramáticas no-LL(k) (gramáticas que no pueden ser reconocidas por analizadores LL) para ningún k.

3.3. Implementación del compilador

Para el sistema implementado, se desarrolló un pequeño lenguaje con el fin de reconocer sistemas de ecuaciones diferenciales, su correspondiente reconocedor y traducción a código Java. Se utilizó ANTLR para su construcción. Esta sección contiene una descripción general del compilador, tanto de su uso como de su implementación.

El lenguaje implementado no intenta revolucionar el mundo de los lenguajes, sino simplemente ser un lenguaje que permita:

- Reconocer sistemas de ecuaciones diferenciales ordinarias de primer orden.
- Definir cualquier número parámetros libres.
- Definir variables algebraicas y las ecuaciones que las definen.
- Insertar comentarios y anotaciones.
- Insertar cualquier cantidad de caracteres de espacio, tabulación y salto de línea, los cuales serán ignorados a partir del nivel léxico. Con el fin de hacer más fácil la lectura.

Este mismo lenguaje trendrá algunas limitaciones, entre ellas:

- No admite compilaciones que involucren a más de un archivo.
- Implementa una recuperación de errores en modo pánico.
- Sólo reconoce diferenciales de primer orden.

3.3.1. Analizador léxico

En esta sección se describen las características y reglas que definen el analizador. Los distintos grupos de *tokens*, que constituyen el lenguaje y que pueden ser utilizados por el usuario para la definición de sistemas dinámicos.

Opciones del analizador

El analizador utiliza un rango de caracteres que está entre el carácter *unicode* 3 y el 377, lo que permite todos los caracteres ASCII, las letras acentuadas (y acentos posibles) y algún símbolo especial de puntuación.

El *lookahead* por defecto del analizador es de dos, con el fin de prescindir de algunos predicados sintácticos que sería muy tedioso escribir.

Palabras reservadas

Las palabras reservadas no pueden ser usadas por los usuario para la definición de variables. Estas palabras reservadas corresponden a las funciones que podrán ser utilizadas para la definición de variables. El lenguaje es sensible a las mayúsculas. Todas las palabras reservadas deberán escribirse con la primera letra en mayúscula y el resto en minúsculas. Las palabras reservadas se presentan en la Tabla 3.1.

Tabla 3.1: Palabras reservadas del lenguaje de definición de sistemas dinámicos

Palabra reservada	Descripción
Log	Nombre de la función logaritmo base 10
Ln	Nombre de la función logaritmo natural
Exp	Nombre de la función exponencial
Sin	Nombre de la función seno
Cos	Nombre de la función coseno
Tan	Nombre de la función tangente
Cot	Nombre de la función cotangente
Sec	Nombre de la función secante
Csc	Nombre de la función cosecante
Sqrt	Nombre de la función raíz cuadrada
Param	Palabra reservada para definir parámetros libres en el sistema

Variables

Una variable a nivel léxico en nuestro lenguaje, es una letra o carácter de subrayado seguidos de una secuencia (que puede ser vacía) de letras, dígitos y caracteres de subrayado,

siempre y cuando no sea una palabra reservada del lenguaje y no empiece por un dígito. En un inicio definiremos dos tipos de variables: las algebraicas y las de estado.

Las variables algebraicas que se utilizan (a nivel sintáctico) tanto en la definición de variables algebraicas como en la definición de variables de estado. Éstas son (como ya se habia mencionado), una letra o carácter de subrayado seguidos de una secuencia (que puede ser vacía) de letras, dígitos y caracteres de subrayado.

```
  Var1= Var2 * 3.1416;
/*Esta linea es la definicion de Var1*/
```

En el ejemplo anterior vemos que Var1 y Var2 son variables algebraicas, puesto que coinciden con la definición léxica dada. Sin embargo, la línea corresponde a la definición de Var1 mientras que Var2 es utilizada para definir Var1 (se encuentra dentro de la definición).

Las varibles de estado son las variables que definen prescisamente el estado del sistema y generalmente se encuentran en expresiones que definen su derivada con respecto al tiempo. A nivel léxico su definición es la misma que la de una variable algebraica, sólo que es necesario que ésta termine con el caracter apóstrofo (’). Como veremos más adelante este tipo de léxico sólo se utiliza para definir una variable de estado.

```
  VarEdo1'= Var2 * VarEdo1;
/*Esta linea es la definicion de la variable de estado VarEdo1*/
```

En el ejemplo anterior vemos que aunque a nivel semántico VarEdo1’ y VarEdo1 son la misma variable (una variable de estado), cuando se utiliza dentro de una definición se utiliza el léxico de una variable algebraica.

Símbolos y operadores

El lenguaje de definición hace uso de un conjunto de símbolos que se utilizan como separadores de distintas unidades semánticas. Estos símbolos se encuentran definidos en la Tabla 3.2.

Tabla 3.2: Símbolo del lenguaje de definición de sistemas dinámicos

Símbolo	Descripción
Paréntesis ( )	Para separar, anidar y cambiar la prioridad de los operadores en expresiones aritméticas.
Corchetes [ ]	Para delimitar el parámetro de una función.
Punto y coma ;	Para delimitar una ecuación.

Los operadores aritméticos son presentados en la Tabla 3.3 cada uno con su prioridad. Los paréntesis se utilizan para cambiar esta prioridad de la forma usual.

Tabla 3.3: Operadores aritméticos del lenguaje de definición de sistemas dinámicos

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación
/	División
^	Potencia
=	Asignación

Enteros y reales

Por supuesto que es necesaria la introducción de números en la definición de sistemas dinámicos. El lenguaje admite números enteros y reales con el mismo formato que en cualquier lenguaje de programacíon.

Comentarios

Es posible especificar dos tipos de comentarios: de una sola línea y de varias líneas. Los comentarios serán totalmente ignorados en los análisis posteriores (sintáctico y semántico), así que son filtrados de la misma forma que los blancos.

```
/* Este es un
   Comentario
   multilinea */

// Este es un comentario de una sola linea
```

Como podemos apreciar, los comentarios se escriben en el formato del lenguaje C. Los comentarios multilínea son agrupaciones de líneas delimitadas por las cadenas "/*" y "*/" y los de una sola línea son una línea indivudial, precedida de la doble barra diagonal "//".

Consideraciones

Cada vez que el usuario inserta, borra o modifica algún caracter (en el panel que contiene la definición del sistema dinámico), se realiza un análisis léxico con el propósito de resaltar los diferentes componentes léxicos del lenguaje y de advertir de la existencia de elementos que no son reconocidos por el lenguaje.

El analizador léxico filtra y elimina tanto los espacios, tabulaciones, saltos y retornos de línea y los comentarios, por lo que fue necesario reconstruirlos manualmente después de cada análisis léxico.

3.3.2. Analizador sintáctico

El análisis sintáctico está relacionado con la definición de las reglas del lenguaje. Como salida de nuestro analizador sintáctico, tendremos un AST. Es decir, el propósito de este análisis es el de la generación de un árbol que represente el sistema dinámico, donde cada uno de los nodos de este árbol sea un *token* definido en el analizador léxico o un *token imaginario* (que se describirá más adelante).

Opciones de analizador sintáctico

Al igual que el analizador léxico, el analizador sintáctico hace uso de un *lookahead* de dos. El analizador sintáctico importa el vocabulario del analizador léxico y exporta su AST, para su posterior utilización por parte del analizador semántico.

Tokens imaginarios

Los tokens imaginarios se utilizan para enraizar los árboles que lo necesiten, normalmente en reglas que no tengan un símbolo apropiado para la raíz. Para nuestro caso es

necesario el uso de tres tokens imaginarios: MODEL, EQUATION y OP_MENOS_UNARIO. MODEL, será la raíz de todo el sistema dinámico, EQUATION la raíz de cada una de la ecuaciones que componen el sistema, y OP_MENOS_UNARIO es la raíz de las expresiones negativas (cuando se utiliza el signo menos unario). Fuera de estos *tokens imaginarios*, los demás nodos del árbol son *tokens* generados por el analizador léxico.

Las reglas

A continuación se definen las reglas sintácticas del lenguaje. Sin embargo las acciones que se realizan para enriquecer el AST generado y las instrucciones de recuperación de errores no son presentadas aquí. La descripción de la sintaxis de un sistema dinámico, se hará de arriba hacia abajo, es decir, viendo el sistema como una estructura árborea, empezaremos por describir la raíz y de ahí se comienza a descender por la estructura hasta llegar a las hojas del árbol.

Un modelo (un sistema dinámico) es una lista de una o más ecuaciones. Donde cada ecuación es la definición de una variable de estado, una variable algebraica o un parámetro libre del sistema.

```
modelo: (ecuacion)+;
```

Una ecuación es una expresión de asignación, una *variable* seguida del signo de asignación (*IGUAL*), seguido de una *expresión aritmética* o de la palabra reservada *Param* (para declarar una variable, como un parámetro libre del sistema) y finaliza con el signo punto y coma (";").

```
ecuacion: variable IGUAL^ (expAritmetica|PARAM) PTO_CMA!;
```

El operador "^ " en la regla anterior indica que el signo de asignación (*IGUAL*) será la raíz del árbol formado por esa regla. Una *variable* es una variable de estado o una variable algebraica.

```
variable: VAR
          |VAR_EDO
          ;
```

Una expresión aritmética puede tener cualquier combinación y número de operaciones aritméticas. Sin embargo, los operadores con menos prioridad y por ende las operaciones que se evalúan al final son la suma y la resta (si es que se encuentran presentes en la expresión), a continuación el producto y el cociente, en seguida de estos la potencia y por último el cambio de signo (que tiene mayor prioridad, por lo que dentro del AST se encuentra más cerca de las hojas y se ejecuta primero). Toda esta jerarquía queda representada en las siguientes reglas:

```
expAritmetica: expProducto ((OP_MAS^|OP_MENOS^) expProducto)*;

expProducto:    expPotencia((OP_PRODUCTO^|OP_COCIENTE^)expPotencia)*;

expPotencia:    expCambioSigno( OP_POTENCIA^ expCambioSigno)*;

expCambioSigno:    (OP_MENOS! raizAcceso
                   | (OP_MAS!)? raizAcceso
                   ;
```

Aún queda por resolver la regla *raizAcceso*. Esta regla es la que reconoce los números y variables. Además también se incluyen tanto los paréntesis, para modificar la jerarquía de los operadores como el acceso a las funciones del lenguaje. El símbolo "!" filtra el token que lo antecede, para que no forme parte del AST.

```
raizAcceso:  VAR
             | NUMERO
             | llamada
             | PARENT_AB! expAritmetica PARENT_CE!
```

A nivel sintáctico, en la parte derecha de todas las asignaciones (ecuaciones) sólo pueden encontrarse variables algebraicas. La cuestión de conocer como ha sido definida cada una de las variables (si como algebraica o como de estado) es realizada en el análisis semántico.

La regla *llamada*, define la sintaxis de la llamadas a las funciones disponibles en el lenguaje. La sintaxis de llamado a una función es: el nombre de la función, seguido de un corchete izquierdo, seguido del parámetro de la función que es en realidad una expresión aritmética y por último un corchete derecho. La regla es:

```
llamada:  funcion CORCH_AB! expAritmetica CORCH_CE!
```

Por último, los nombres de las funciones permitidas son los mostrados en la Tabla 3.1.

Para ver un ejemplo práctico de la sintaxis, suponga el sistema que representa el comportamiento de un oscilador lineal:

$$\begin{aligned} \dot{x_1} &= x_2 \\ \dot{x_2} &= -w^2 x_1 - 2\mu x_2 + F cos(\Omega) \end{aligned} \tag{3.1}$$

donde w representa un parámetro libre del sistema, mientras que $\mu = 2$, $F = 10$ y $\Omega = 2$. La representación de este sistema en el lenguaje de definición sería la siguiente:

```
x1' = x2;
x2' = -w^2*x1 - 2*Mu*x2 + F*Cos[Omega];
w = Param;
F = 10;
Mu = 2;
Omega = 2;
```

El analizador sintáctico genera el AST presentado en la Figura 3.3.

Consideraciones

A diferencia del análisis léxico, el análisis sintáctico es realizado una vez que el usuario intenta ingresar su modelo al sistema. Si el análisis no es completado de manera satisfactoria, no se efectua el análisis semántico. Es decir, que el análisis sintáctico no es realizado en *línea* como se realiza el análisis léxico.

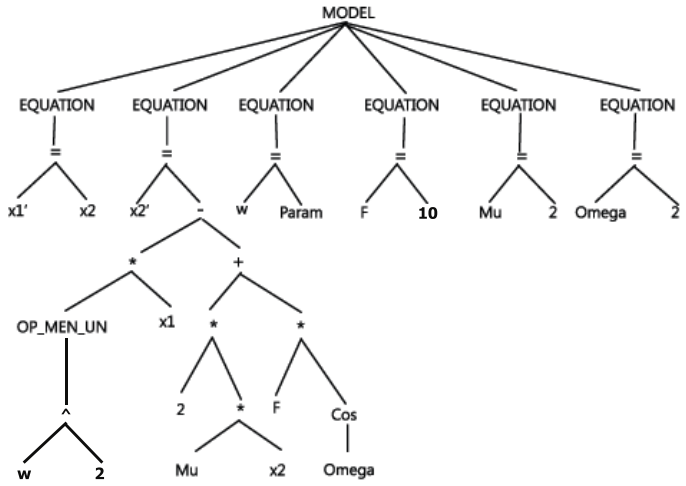


Figura 3.3: AST generado por el analizador sintáctico

3.3.3. Recuperación de errores

La recuperación de errores no es una fase más del proceso de compilación. En lugar de eso, debería considerarse como una característica deseable del proceso de compilación.

La recuperación de errores permite al compilador detectar un error, dar parte de él y seguir reconociendo la entrada, detectando de esta manera más errores. Si el programador puede ver más errores en cada compilación, podrá corregir más errores en cada compilación, por lo que será más eficiente. En general la recuperación de errores tiene tres fases básicas:

1. Detección. El error se localiza.

2. Informe. Se archiva o se muestra un mensaje informativo explicando la naturaleza del error.

3. Recuperación. El analizador utiliza alguna técnica para superar el error y poder seguir efectuando el análisis.

ANTLR se encarga de realizar la detección y de generar el mesaje de error correspondiente. La tarea de mostrar dicho mensaje es un asunto del programador. En cuanto a la recuperación, ANTLR permite implementar distintas técnicas de este tipo.

En general, cualquier error de reconocimiento léxico o sintáctico en ANTLR implica el lanzamiento de una excepción. Por lo que bastará con añadir manejadores de excepciones para hacer un buen manejo de errores.

La recuperación de errores consiste en resincronizar la entrada con respecto a las reglas, de manera que se pueda seguir con el análisis. La estrategia que utiliza ANTLR por defecto, es la de descartar tokens de la entrada hasta que se encuentre un token válido que permita sincronizar.

Dada una expresión de reconocimiento (que puede ser una regla, una alternativa o una subregla EBNF), se define el conjunto *SIGUIENTE (expresión)* como el conjunto de tokens que pueden seguir a dicha expresión en la gramática en la que se encuentra. Por ejemplo, en la gramática:

```
base : IDENT
       | LIT_ENTERO
       | LIT_REAL
       | LIT_CADENA
       ;
expresion : exprSuma ;
exprSuma : e1:exprProducto (OP_SUMA exprProducto)* ;
exprProducto : e2:base (OP_PRODUCTO base)* ;
```

Podemos calcular varios conjuntos *SIGUIENTE* en diferentes lugares (que se marcan con etiquetas):

- *SIGUIENTE (exprSuma-la regla)*: Es *OP_SUMA* o el fin del archivo.
- *SIGUIENTE (e1)*: Es *OP_SUMA* o el fin del archivo.
- *SIGUIENTE (e2)*: Es *OP_PRODUCTO* o el fin del archivo.

Una vez detectado el error, se consumen tokens hasta que se encuentre alguno coincidente con el conjunto *SIGUIENTE* de la regla en la que se ha detectado. Los pasos a seguir son exactamente tres:

- Mostrar el error al usuario (con una llamada al método *reportError*).
- Consumir el token que ha provocado el error (llamando a *consume*).
- Consumir tokens de la entrada hasta que alguno sea válido (pertenezca al conjunto *SIGUIENTE (regla)*, donde regla es la regla en la que se ha producido el error). Todo ello se realiza con una simple llamada al método *consumeUntil*, pasándole como parámetro el conjunto *SIGUIENTE (regla)*.

A esta estrategia utilizada por ANTLR, se le denomina *modo pánico*.

3.3.4. Análisis semántico

El análisis semántico tiene como propósito detectar los errores semánticos (incoherencias con respecto a las reglas semánticas del lenguaje) y eriquecer la información semántica del AST que será necesaria para la generación de código.

El principal objetivo del proceso del análisis semántico es la construcción de una estructura llamada *lista semántica*, la cual es utilizada para validar la semántica del sistema.

Esta estructura es una lista ligada, donde cada nodo de la lista es la cabecera de otra lista ligada. Cada una de las cabeceras representan una variable (algebraica o de estado) o un parámetro libre del sistema, mientras que cada nodo de la lista que encabezan representa una variable o parámetro libre que forma parte de la definición de la cabecera (de la variable que representa).

Para el caso del sistema definido en el sistema 3.1, la lista semántica generada sería la mostrada en la Figura 3.4 Cada nodo de la lista a su vez también es un objeto que contiene:

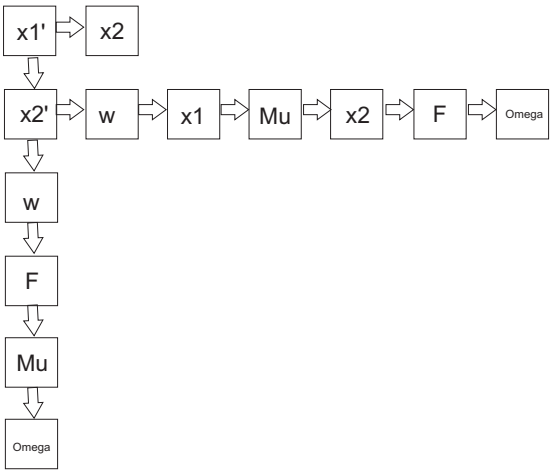


Figura 3.4: Lista semántica del sistema 3.1

- Una cadena, que es la variable o parárametro que representa el nodo.
- Un entero, que identifica el tipo de variable (algebraica, de estado o parámetro).
- Un entero que contendrá la posición de la ecuación representada por el nodo (cabecera), dentro del código o *script* generado en la siguiente etapa.
- Un entero que contiene la posición de la ecuación representada por el nodo (cabecera), dentro del documento de la interfaz.
- Dos enteros que representan la línea y columna que tiene la variable dentro del documento de la interfaz.
- Un dato booleano que se utiliza para conocer si una variable ha pasado ya por un proceso de verificación.

La lista semántica implementa una serie de métodos que tienen como propósito verificar la semántica del sistema. Los puntos semánticos que se analizan son:

- Que cada una de las variables haya sido declarada.
- Que no haya duplicidad de variables.
- Que no existan bulcles de inicialización.

Cabe destacar que el usuario puede realizar todas las declaraciones de variables y parámetros en el orden que desee. Es decir, que no es necesario que todas las variables utilizadas en una ecuación hayan sido declaradas en líneas anteriores. Pueden declararse en líneas posteriores. Es trabajo del análizador semántico y del generador de código reordenar el conjunto de definiciones, de tal manera que cuando se genere el código Java correspondiente al sistema, todas las variables utilizadas hayan sido declaradas con anterioridad.

El análisis semántico es realizado una vez que no existen errores sintácticos, por lo que si existen errores sintácticos y semánticos en la definición del sistema, el compilador sólo nos mostrará los errores sintácticos y sólo hasta que éstos hayan sido depurados, se procede con el análisis semántico y los errores de este tipo serán mostrados.

3.3.5. Generación de código

La generación de código es la última fase del proceso de compilación (Figura 3.5) y es realizada una vez que no existe error de ningún tipo en la entrada.

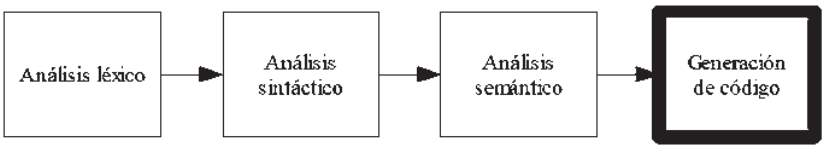


Figura 3.5: Última fase del proceso de compilación: Generación de código

La generación de código toma como entrada una representación abstracta del código fuente y produce como resultado un código equivalente. Para nuestro caso, el objetivo de la fase de generación de código es el de la construcción de un arreglo de cadenas (*Strings*) que representan parte de una serie de instrucciones Java. Dicho arreglo será parte de la función objetivo utilizada por alguna metaheurística en la búsqueda de puntos fijos.

La generación de código hace uso del mismo principio utilizado para realizar el análisis semántico. Se realiza un recorrido del AST y con la ayuda de una pila y de la *lista*

semántica, se va construyendo y ordenando (ya que a diferencia del lenguaje de definición el código objeto si requiere que las instrucciones tengan un orden bien definido) cada una de las instucciones en código Java.

La estructura de esta función o de este conjunto de instrucciones se divide en cuatro partes:

- Definición e inicialización de las variables de estado.
- Definición e inicialización de los parámetros libres del sistema.
- Definición y evaluación de las variables aritméticas del sistema.
- Evaluación de las variables de estado.

Este conjunto de cadenas es un *script* incompleto, que forma parte de un objeto del cual se habla en el próximo capítulo, llamado *SystemFunction*. Es incompleto debido a que los valores inciales de las variables de estado y los parámetros son asignados dinámicamente y corresponden a los valores con los que se desee evaluar la función. El *script* generado para el sistema 3.1, se vería como se muestra en la Tabla 3.4

Tabla 3.4: Script del sistema 3.1 generado por el analizador

”double x2 =”
”double x1 =”
”double w =”
”double F = 10;”
”double Mu = 2;”
”double Omega = 2;”
”= -w * x1 - 2 * Mu * x2 + F * Math.cos(Omega);”
”= x2;”

El conjunto de sentencias generadas (correspondientes al Sistema 2.1) se ejecutarían en el orden que se muestran en la Tabla 3.4, mediante el uso de una librería de scripting Java, de la cual se comentará en el siguiente capítulo.

3.4. Comentarios finales

Hasta aquí se describió la implementación de las distintas fases del compilador y reconocedor de sistemas de ecuaciones diferenciales. El cual tiene las características necesarias que permiten ingresar al sistema con facilidad modelos que representen sistemas dinámicos y que posteriormente puedan ser analizados a través de diagramas de bifurcación.

El análisis léxico, se realiza en tiempo real (mientras el usuario se encuentra introduciendo la definición del sistema) y se encarga de reconocer los *tokens* que conforman el lenguaje de definición y resaltarlos según su tipo (operadores, operandos, variables de estado, etc.). El analizador sintáctico, valida la estructura del conjunto de ecuaciones del modelo y construye un AST (Abstract Syntax Tree) que es utilizado en posteriores fases del proceso de compilación. El analizador semántico verifica cuestiones como la declaración e inicialización de variables. Durante este análisis se construye una estructura llamada *lista semántica*, la cual es una lista ligada "vertical" y "horizontalmente" que es utilizada tanto para la verificación semántica como para el reordenamiento del conjunto de ecuaciones, de tal manera que las sentencias en el código objeto puedan ser ejecutadas por la máquina virtual de Java. La generación de código, se lleva a cabo mediante el uso del AST, la *lista semántica* y una pila que funciona como recipiente para operandos y representa la parte de cada ecuación que se ha construido.

El compilador genera un *script* en lenguaje Java, que recibe los valores iniciales de las variables de estado y de los parámetros, evalua el sistema y regresa el vector de estado del sistema.

Capítulo 4

Cálculo de puntos fijos

En el presente capítulo se describe la metaheurística incorporada por defecto, utilizada para la búsqueda de puntos fijos de un sistema dinámico. También se describe el mecanismo para la incorporación y ejecución de los módulos de búsqueda de puntos de equilibrio.

Como ya se ha mencionado, el problema de la búsqueda de puntos fijos en un sistema dinámico puede ser planteado como un problema de optimización. De esto se desprende la idea de resolverlo mediante técnicas metaheurísticas. Dado que una de las intenciones principales de la implementación es la de la construcción de una plataforma que permita la utilización de distintas técnicas, es necesario brindar una opción por defecto. Así que se ha implementado un algoritmo de enjambre de partículas (PSO, por sus siglas en inglés *Particle Swarm Optimization*) con nichos como técnica por defecto en la búsqueda de puntos fijos. La implementación es un trabajo propuesto por Julio A. Barrera Mendoza Barrera en su tesis doctoral [Barrera08]. A continuación se describe dicho algoritmo.

4.1. Optimización de enjambre de partículas

El algoritmo de optimización de enjambre de partículas es un algoritmo inspirado en la naturaleza. Este algoritmo está basado en la observación del comportamiento social de los individuos en una especie; por ejemplo los cardúmenes de peces y las parvadas de

aves. Cuando un individuo encuentra una región con alimento los demás lo seguirán.

Este algoritmo comparte muchas de las características de los algoritmos genéticos: Comienza con una población inicial, cada partícula también representa un punto en el espacio de búsqueda y de igual manera, cada partícula posee un valor determinado por la función que se quiere optimizar; este valor es utilizado para dirigir el movimiento de las partículas y representa la cantidad de alimento en la posición donde se encuentra la partícula. Una diferencia con algoritmos genéticos es que no se descartan los individuos de la población inicial, estos sólo actualizan su posición. Cada partícula conoce dos elementos de información: el primero corresponde a la posición de la partícula con el valor máximo (componente social); el segundo corresponde a la posición donde la partícula ha obtenido el mayor valor (componente cognitivo). El nombre de partículas fue dado por lo autores del algoritmo debido que se usa el concepto de posición y velocidad, así es mejor hablar de la posición y velocidad de una partícula más que de un individuo.

De la misma forma que los algoritmos genéticos, en la optimización de enjambre de partículas se lleva a cabo un proceso iterativo para encontrar el óptimo de una función determinada. Se inicia con un enjambre inicial, una colección de partículas colocadas aleatoriamente dentro del espacio de búsqueda. Una vez generado el enjambre inicial cada partícula es evaluada; la partícula con el valor máximo (o mínimo según sea el problema) es seleccionada y la posición de esta partícula es tomada en cuenta por cada una de las partículas. Después de la evaluación se calcula una velocidad, considerando los dos componentes de información guardados en la partícula. Se calcula una velocidad hacia la mejor partícula y otra hacia la posición del mejor valor registrado.

En cada iteración se selecciona la partícula con mejor valor, así la posición a donde se dirigen la partículas no siempre es la misma. De la misma forma en cada iteración se actualiza la posición del mejor valor registrado por la partícula; de esta forma los dos componentes de información de la partícula siempre están cambiando.

4.1.1. Implementación del algoritmo de enjambre de partículas

El algoritmo de optimización de enjambre de partículas fue implementado utilizando un arreglo de vectores de números de punto flotante. Cabe señalar que dado que el algoritmo de enjambre de partículas utiliza los conceptos de velocidad y distancia, es necesario que la posición de las partículas sea en un espacio donde tengamos una función de distancia bien definida [Haupt04].

Una vez generado, inicializado y evaluado un enjambre inicial, la velocidad y posición de todas las partćulas es actualizada de acuerdo a las reglas dadas en las Ecuaciones 4.1 y 4.2.

$$v_{new} = \chi[v + C_1R_1(P_{best} - P) + C_2R_2(P_{global} - P)] \quad (4.1)$$

$$P_{new} = P + V_{new} \quad (4.2)$$

donde v y P son la velocidad y posición actual, respectivamente; R_1 y R_2 son números aleatorios generados en el rango de [0, 1]; C_1 y C_2 son los factores de aprendizaje; P_{best} es la posición de la partícula con mejor valor de aptitud registrada hasta la iteración actual; P_{global} es la posición de la partícula con el mejor valor de aptitud en el enjambre; v_{new} y P_{new} son la nueva velocidad y posición de la partćula. La constante χ es calculada de acuerdo a las Ecuaciones 4.3 y 4.4.

$$\phi = C_1 + C_2 \quad (4.3)$$

$$\chi = \frac{2}{|2 - \phi - \sqrt{\phi^2 - 4\phi}|} \quad (4.4)$$

La constante χ llamada coeficiente de constricción [Kennedy J.01, Clerc02, Li X.06] evita que las partículas exploren demasiado lejos del espacio de búsqueda. Así no necesitamos una constante v_{max} para limitar la velocidad de las partículas [Li X.06].

El trabajo propuesto en [Li X.06], agrega dos pasos al método base de optimización de enjambre de partículas; después de que el enjambre inicial es creado, la partícula con el mejor valor de aptitud es seleccionada y todas las partículas con una distancia menor que el valor de un parámetro r son seleccionadas como individuos de una especie. El óptimo

global de las partículas en la especie es evaluado con la partícula inicialmente seleccionada. Las partículas en la especie son marcadas como usadas y la siguiente partícula con el mejor valor de aptitud es seleccionada para crear una nueva especie. Este procedimiento es repetido hasta que todas las partículas estén dentro de una especie (el valor de aptitud de todas las partículas permanece sin alteraciones). Después de que todas las especies han sido formadas sólo un número de partículas debe permanecer en cada especie, sólo las n partículas más cercanas a la partícula con el mejor valor de aptitud son conservadas dentro de la especie, las partículas restantes son reinicializadas en posiciones aleatorias. El Algoritmo 4.1.1 muestra la optimización de enjambre partículas con especies.

Algoritmo 1 Enjambre_particulas_especies()

```
1: generar enjambre
2: evaluación inicial
3: for i ← 1 to número_iteraciones do
4:     calcular especies
5:     calcular velocidades
6:     actualizar partículas
7:     reiniciar partículas rechazadas
8: end for
```

En el sistema desarrollado, el algortimo es ejecutado para cada valor o par de valores de los parámetros libres que formen parte del diagrama de bifurcación. Es decir, que cada vez que algún parámetro cambia de valor, un experimento independiente es realizado utilizando alguna metaheurística (por defecto optimización de enjambre de partículas). Este procedimiento es ampliado en el siguiente capítulo.

Función de mapeo

El trabajo presentado en [Barrera08] implementa una propuesta hecha en [Aggarwal00], la cual consiste en una transformación para convertir el problema de búsqueda de soluciones (puntos x para los cuales $f(x) = 0$), en una búsqueda de máximos. Ésto se logra mediante la aplicación de la ecuación 4.5

$$g(x) = \frac{1}{1 + |f(x)|} \tag{4.5}$$

Este mapeo se usó en conjunto con el algoritmo genético de tipo niching desarrollado por Alan Petrowski [Petrowski97], en el que se divide a la población en subpoblaciones y dónde sólo el individuo con la aptitud máxima de la subpoblación es candidato a reproducción.

De la ecuación 4.5 se observa que para x tal que $f(x) = 0$ el valor de $g(x)$ es de uno. Para cualquier otro valor de $f(x)$ diferente de cero el valor de $g(x)$ será menor de uno, pero sin llegar a ser cero, es decir, $g(x) \to 0$ cuando $x \to \pm\infty$. Ésto es, el problema de buscar soluciones para $f(x)$ se convierte en la búsqueda de valores máximos de $g(x)$.

4.2. Adición de nuevos módulos

El módulo de cálculo de puntos de equilibrio de un sistema no forma parte de la aplicación desarrollada. Es decir, el módulo de enjambre de partículas del cual se habló en la sección anterior, fue desarrollado como se tendría que desarrollar un nuevo módulo que implemente una metaheurística distinta. En esta sección se describe el proceso de implementación de una metaheurística que sea utilizada por el sistema para el cálculo de puntos fijos.

4.2.1. La clase MetaHeuristic

Para que un módulo (que implemente una metaheurística) pueda ser utilizado por el sistema desarrollado, el módulo debe extender la clase *MetaHeuristic*. La clase *MetaHeuristic* es una clase abstracta, la cual define los miembros y métodos necesarios para la implementación de una metaheurística. La clase *MetaHeuristic* define los miembros:

```
private SystemFunction objectiveFunction;

private double [][] ranges;
```

El objeto *SystemFunction* encapsula el *script* que representa un sistema dinámico (la función objetivo). El programador sólo tendrá que hacer una llamada al método *evalSystem* que recibe como parámetros:

- Un arreglo de flotantes dobles que contiene los valores de las variables de estado del sistema (el vector de estado). Este arreglo podría representar un cromosoma (en el caso de algoritmos genéticos) o la posición una partícula (en el caso de PSO).
- Un arreglo de flotantes dobles que contiene los valores de los parámetros libres (como ya se mencionó, se ejecuta un experimento independiente para un valor o par de valores de los parámetros).

El arreglo bidimensional *ranges* contiene los rangos de operación de cada una de las varibles de estado del sistema dinámico. El usuario (programador) sólo tendrá que hacer uso de este arreglo para la generación de sus individuos o elementos en su metaheurística.

La clase *MetaHeuristic* define los métodos:

```
public abstract double [][] findFixedPoint(double [] freeParams)
throws Exception;

public abstract String [][] getMetaParamsName();

public abstract String getMetaHuristicDescription();

public abstract void setMetaParams(String []p);
```

El método *findFixedPoint* es el método que la aplicación llama para encontrar los puntos fijos dado un conjunto de valores correspondientes a los parámetros libres del sistema, contenidos en el parámetro *freeParams*, el cual es un arreglo bidimensional de flotantes dobles. Este método debe regresar un arreglo de flotantes dobles que contenga los puntos de equilibrio para los valores de parámetros definidos en el argumento *freeParams*. Cada una de las filas de este arreglo representa un punto de equilibrio, mientras que las columnas representan los valores para cada una de las variables de estado del punto fijo. Este método lanza una excepción, de tal manera que sea posible informar a la aplicación de la existencia de algún error.

El método *getMetaParamsName* es invocado por la aplicación con el fin de obtener los nombres y valores por defecto que utiliza la metaheurística y mostrarlos al usuario; no recibe parámetros de entrada y tiene como valor de retorno un arreglo bidimensional de cadenas (*Strings*). Este arreglo debería tener dos columnas, de esta manera, cada fila es un par de valores *nombre de parámetro - valor por defecto*

El método *getMetaHeuristicDescription* no recibe parámetros de entrada y tiene como valor de retorno una cadena que contiene la descripción de la metaheurística y sus parámetros. Es responsabilidad del programador darle formato a dicha cadena.

El método *setMetaParams* es utilizado para pasar al módulo los valores de los parámetros de la metaheurística. Recibe como parámetro de entrada un arreglo de cadenas que contiene los valores de cada uno de los parámetros seleccionados por el usuario para ejecutar la metaheurística. Es responsabilidad del programador validar y convertir a los tipos de datos adecuados. Los valores dentro del arreglo son pasados en el mismo orden en el que el programador los definió en el método *getMetaParams*.

De esta manera, un programador que desee implementar una metaheurística para que sea incorporada al sistema, debe de extender la clase *MetaHeuristic*. Para ello debe de hacer uso del archivo *metaLib.jar* e importar a su archivo fuente las clases: *Inter.MetaHeuristic* e *Inter.SystemFunction*.

Es importante comentar que para que el módulo que se añada funcione, deben de encontrarse los archivos *.class* del módulo y de todos los demás archivos del que éste haga uso en el mismo directorio y paquete Java (*package*).

4.2.2. Programación reflexiva

La reflexión (o reflexión computacional) es la capacidad que tiene un programa para observar y opcionalmente modificar su estructura de alto nivel [Valbuena08]. Normalmente, la reflexión es dinámica o en tiempo de ejecución, aunque algunos lenguajes de programación permiten reflexión estática o en tiempo de compilación [Valbuena08]. Es más

común en lenguajes de programación de alto nivel ejecutándose sobre una máquina virtual, como Smalltalk o Java y menos común en lenguajes de programación de bajo nivel como C.

En un sentido más amplio, la reflexión es una actividad computacional que razona sobre su propia estructura. Cuando el código fuente de un programa se compila, normalmente se pierde la información sobre la estructura del programa conforme se genera el código de bajo nivel (normalmente lenguaje ensamblador). Si un sistema permite reflexión, se preserva la estructura como metadatos en el código generado. Dependiendo de la implementación, el código con reflexión tiende a ser más lento que el que no lo tiene.

Un lenguaje con reflexión proporciona un conjunto de características disponibles en tiempo de ejecución que de otro modo serían muy difícilmente realizables en un lenguaje de más bajo nivel. Algunas de estas características son las habilidades para:

- Descubrir y modificar construcciones de código fuente (tales como bloques de código, clases, métodos, protocolos, etc.) como objetos de categoría superior en tiempo de ejecución.
- Convertir una cadena que corresponde al nombre simbólico de una clase o función en una referencia o invocación a esa clase o función.
- Evaluar una cadena como si fuera una sentencia de código fuente en tiempo de ejecución.

Ejecución de módulos de búsqueda de puntos fijos - Java Reflection

Este apartado describe la forma en la que se logra que un módulo que implemente una metaheurística para la búsqueda de puntos fijos sea ejecutada sin haber sido compilada en el mismo proyecto que la aplicación principal. Además se describirán de manera general las herramientas utilizadas.

El API (Application Programming Interface) Reflection de Java es la herramienta que nos permite realizar programación reflexiva en Java [Valbuena08]. Sin embargo, a

pesar de su potencial, es un API un tanto desconocido por muchos programadores. Reflection es un API que puede describir cualquier objeto aún cuando este objeto no haya sido creado por nosotros (o al menos no dentro del mismo proyecto).

Todos los objetos en Java heredan de la clase *java.lang.Object* y por ello están dotados de un método *getClass*, cuya definición es *public final Class getClass()*. Este método devuelve un objeto *java.lang.Class*, que es el punto de entrada al API Reflection y permite cargar clases dinámicante a través su método *forName* que se utiliza para cargar e instanciar clases del *classpath*.

De esta manera, para verificar si una clase (el archivo *class*) es en realidad un módulo para el cálculo de puntos fijos útil para nuestra aplicación, el primer paso es cargar dinámicamente la clase candidata mediante un *ClassLoader*, que es un objeto de *Java Reflection* que permite cargar clases en memoria bajo demanda a través del método *forName*. Una vez cargada una clase, es necesario verificar si ésta extiende la clase abstracta *MetaHeuristic*. Para esto se utiliza el método *getSuperclass*.

Usando Java Reflection, la forma de invocar constructores y métodos de clases cargadas dinámicamente es diferente a la forma usual. Para ello se utilizan los métodos:

```
 java.lang.reflect.Method getMethod(String name, Class[] parameterTypes)
 java.lang.reflect.Constructor getConstructor(Class[] parameterTypes)
Object invoke(Object obj, Object[] args)
<T> newInstance(Object [] initargs)
```

El método *getMethod()* devuelve un método público de la clase, a partir de su nombre y de un arreglo con las clases a las que pertenecen cada uno de los parámetros del método. El método *getConstructor* obtiene el constructor público de la clase sobre la que se invoca a partir de un arreglo con las clases a las que pertenecen cada uno los parámetros del constructor. El método *invoke* ejecuta el método sobre un objeto, pasándole los parámetros necesarios y devuelve su resultado. El método *newInstance* usa el objeto constructor sobre

el cual es invocado para crear e inicializar una nueva instancia de la clase a la que pertenece el constructor. Recibe como parámetro un arreglo con los objetos de inicialización.

De esta manera tenemos acceso a información interna de la clases cargadas en la máquina virtual de Java y es posible escribir código que funciona con clases seleccionadas en tiempo de ejecución y que no forman parte de nuestro código fuente.

4.3. Evaluación dinámica de modelos - BeanShell

Esta sección describe la ejecución de código Java en forma dinámica, es decir, evaluar los sistemas dinámicos introducidos por el usuario requiere de ejecutar código que no se encuentra en ningún archivo fuente y que puede ser modificado. Para ello fue necesario ejecutar código Java en modo *scripting*, haciendo uso de *BeanShell*.

BeanShell es un intérprete pequeño, libre e integrable (a código fuente Java) con características de lenguaje *scripting*. *BeanShell* ejecuta instrucciones y expresiones estandar Java. Haciendo uso de esta herramienta es posible ejecutar código Java de forma dinámica en tiempo de ejecución, proporcionándole extensibilidad a nuestra aplicación. Debido a que *BeanShell* está escrito en Java y se ejecuta en la misma máquina virtual que la aplicación que lo invoca, es posible pasar libremente referencias y objetos en secuencias de comandos y devolverlos como resultados a la aplicación.

Este proceso de interpretación es muy sencillo. La clase *Interpreter* implementa precisamente la funcionalidad de un intérprete en modo *scripting*. Una instancia de *Interpreter* es usada para evaluar instrucciones o expresiones Java en modo *script*. El objeto reserva su propio espacio de datos por lo que es posible declarar, recuperar y modificar datos. La clase cuenta con una gran diversidad de métodos, sin embargo para los fines aquí requeridos se utilizan dos:

```
java.lang.Object eval(String statements)

java.lang.Object get(String name)
```

El método *eval* es utilizado para evaluar cualquier instrucción proveniente de un flujo de entrada o de una cadena. Mientras que la recuperación de cualquier elemento (variable u objeto) del intérprete se hace mediante la invocación del método *get*, que recibe como parámetro una cadena que representa el nombre del objeto o variable que se desea obtener. Este método regresa una instancia de la clase *Object*, por lo que es responsabilidad del programador hacer el *casting* correspondiente.

Ejecución de la clase SystemFunction

Se comentó en secciones anteriores acerca de la clase *SystemFunction*. Esta clase encapsula el *script* que es a su vez la función objetivo o de aptitud del proceso de optimización. El programador necesita hacer uso del objeto *objectiveFunction* que es miembro de la clase *MetaHeuristic* y que es una instancia de la clase *SystemFunction*. Para evaluar una posible solución, el programador deberá invocar el método *evalSystem*

```
public double[] evalSystem(double []stateVars,double []parameters)
throws Exception
```

Este método recibe como parámetros la posible solución (los valores de las variables de estado o vector de estado) en el arreglo *stateVars* y los valores de los parámetros libres en el arreglo *parameters*.

El método *evalSystem* sigue el procedimiento mostrado en el Algoritmo 2. Como se comentó en el capítulo anterior, la evaluación de la función objetivo se divide en cuatro partes, cada una de las cuales corresponde con una de las fases del algoritmo.

- Definición e inicialización de las variables de estado.
- Definición e inicialización de los parámetros libres del sistema.
- Definición y evaluación de las variables aritméticas del sistema.
- Evaluación de las variables de estado.

En la primera fase apreciamos que se concatena la línea del *script* con el valor de la variable de estado, esto es debido a que el compilador sólo generó la parte izquierda de la definición y asignación de estas variables, con el objeto de que la parte derecha de la asignación sea proporcionada por el módulo de búsqueda de puntos fijos (valores distintos para distintas posibles soluciones).

La segunda fase es muy similar a la primera, sin embargo, ésta es la encargada de definir e inicializar los parámetros libres (o de bifurcación) del sistema. Al igual que en el caso anterior, el compilador sólo genera la parte izquierda de la asignación.

La tercera fase es la correspondiente a las variables algebraicas y como se puede apreciar, no se hacen concatenaciones. Esto es debido a que el compilador genera la instrucción completa que define e inicializa la variable algebraica correspondiente.

La última fase corresponde a la evaluación misma de una posible solución para el sistema dado. Y corresponde a evaluar el sistema de ecuaciones algebraicas del sistema de ecuaciones diferenciales, sustituyendo todos los valores con los que ya contamos (variables de estado, parámetros de bifurcación y variables algebraicas). Se concatena cada celda del vector de estado con la línea del *script* correspondiente. Debido a que el compilador generó la expresión correspondiente a la parte derecha de la ecuación diferencial. Entonces lo que realmente se está realizando aquí es evaluar esa expresión y asignarla a la variable de estado correspondiente.

4.4. Comentarios finales

La búsqueda de puntos fijos se realiza mediante un módulo que implementa un algoritmo de enjambre de partículas. Dicho algoritmo evoca el comportamiento de ciertos grupos de animales para encontrar alimento. Tiene similitudes con los algoritmos genéticos, sin embargo, una de las pricipales diferencias es que los individuos no se desechan, si no que sólo se mueven a través del espacio de búsqueda. Cada uno de estos individuos posee una posición y una velocidad. Para moverse, cada uno de los individuos toma en cuenta

Algoritmo 2 Evaluación de función de aptitud (*stateVars*, *parameters*

```
1: instanciar intérprete
2: en intérprete inicializar el vector de aptitud
3: lineaScript ← 1
4: /*declaración e inicialización de variables de estado*/
5: for i ← 1 to número de variables de estado do
6:    Script[lineaScript] ← Script[lineaScript]+StateVars[i]
7:    interpretar Script[lineaScript]
8:    lineaScript ← lineaScript + 1
9: end for
10: /*declaración e inicialización de parámetros*/
11: for i ← 1 to número de parámetros do
12:    Script[lineaScript] ← Script[lineaScript]+parameters[i]
13:    interpretar Script[lineaScript]
14:    lineaScript ← lineaScript + 1
15: end for
16: /*declaración y evaluación de variables algebraicas*/
17: for i ← 1 to número de variables algebraicas do
18:    interpretar Script[lineaScript]
19:    lineaScript ← lineaScript + 1
20: end for
21: /*declaración y evaluación de variables de estado*/
22: for i ← 1 to número de variables de estado do
23:    Script[lineaScript] ← vector de aptitud [i] + Script[lineaScript]
24:    interpretar Script[lineaScript]
25:    lineaScript ← lineaScript + 1
26: end for
27: obtener del intérprete el vector de aptitud
28: retornar vector de aptitud
```

la posición de lider y su propia mejor posición. Este algoritmo tiene una variante llamada *PSO con nichos* la cual es más eficiente en problemas multimodales (como es el caso de la búsqueda de puntos fijos).

El sistema desarrollado utiliza *programación reflexiva* para hacer posible la incorporación y utilización de módulos (de búsqueda de puntos fijos) desarrollados de manera independiente al sistema. Un desarrollador sólo requiere: 1) que su módulo esté desarrollado en Java y 2) que el módulo extienda la clase abstracta *MetaHeuristic*. La clase *MetaHeuristic* define los métodos necesarios para que el sistema obtenga los nombres y valores de los parámetros que utiliza la metaheurística, con el fin de obtener los puntos fijos en un determinado rango de operación del sistema.

Para poder evaluar los sistemas introducidos por el usuario, el sistema debe de ejecutar el *script* generado por el compilador de manera dinámica, ya que éste no se encuentra en ningún archivo fuente Java. Para esto el sistema hace uso de un intérprete de Java llamado *beanShell* y de un algoritmo que realiza la evaluación del sistema dinámico.

Hasta aquí hemos descrito el mecanismo de obtención de puntos fijos. Tenemos el conjunto de puntos de equilibrio del sistema en el espacio de búsqueda definido por el usuario. Este conjunto constituye el diagrama de bifurcación, sólo resta trazarlo.

Capítulo 5

Trazo de diagramas

En este capítulo se describe el método general, la implementación y las herramientas utilizadas para lograr el trazo de los diagramas de bifurcación. Además se muestran algunos de los resultados obtenidos mediante el sistema desarrollado.

5.1. Método de trazo de diagramas utilizando metaheurísticas

Dado el conjunto de ecuaciones diferenciales que modela un sistema dinámico con un solo parámetro

$$\dot{x} = f(x, \alpha), x \in R^n, \alpha \in R^1 \tag{5.1}$$

La forma estándar de generar un diagrama de bifurcación utilizando un método tradicional consiste en encontrar un punto fijo estable, y después, usando un método de continuación se determina el siguiente punto fijo, basado en la condición

$$f(x, \alpha) = 0 \tag{5.2}$$

la cual define una curva unidimensional suave en R^{n+1}. Nuestra aproximación a este problema es determinar todos los puntos $x \in X$ para una región dada $X \subset R^n$ que satisfacen la Ecuación 5.2 para un valor dado del parámetro.

Para encontrar todos los puntos x, usando alguna metaheurística (algoritmo de optimización de enjambre de partículas, en primera instancia), primero se transforma el problema de encontrar todas las soluciones para la Ecuación 5.2 en un problema de encontrar todos los máximos usando la transformación dada por la Ecuación 4.5.

Comenzando con un valor para el parámetro libre $\alpha = \alpha_0$, una metaheurística determinada es aplicada un determinado número de iteraciones para encontrar todos los máximos de la función g con el valor fijo α_0. Una vez que son encontrados todos los puntos fijos x para una región dada y parámetro fijo α_0, estos son almacenados, entonces el parámetro α es cambiado a $\alpha_1 = \alpha_0 + \Delta\alpha$ y se determina un nuevo conjunto de puntos fijos x. Δ es un incremento constante aplicado al parámetro y que en la aplicación es llamado *plot step*. A un mayor valor de Δ menor definición en el diagrama y menor carga de trabajo; a un menor valor de Δ mayor definición del diagrama y mayor carga de trabajo. Es decir, este parámetro determina el número de experimentos realizados.

Una nueva búsqueda podría estar basada en la información acumulada del experimento anterior o se podría realizar un experimento nuevo independiente. Basados en lo dicho anteriormente, se genera el diagrama de bifurcación cambiando el parámetro y almacenando todas las soluciones encontradas para cada valor α_k del parámetro.

La estabilidad de los puntos obtenidos no es considerada aún en el sistema y es un asunto considerado para trabajo futuro.

5.2. Procedimiento del trazo del diagrama de bifurcación

Para generar un diagrama de bifurcación por medio de la herramienta desarrollada, el usario debe elegir una variable de estado del sistema y uno o dos parámetros libres del sistema. Es decir, en un diagrama de bifurcación se representa el comportamiento de una de las variables de estado con respecto a la variación de uno o dos parámetros libres del sistema. En la definición de un sistema el usuario puede declarar el número de parámetros

libres que desee, sin embargo, por cuestiones geométricas, sólo se pueden representar dos parámetros como máximo en un diagrama.

El usuario también debe definir (y por supuesto conocer) los rangos de operación de cada una de las variables de estado del sistema. El usuario también debe definir el rango de cada uno de los parámetros que formen parte del diagrama y debe proporcionar los valores de los parámetros libres del sistema que no formarán parte del diagrama. A estos últimos les llamamos parámetros libres estáticos. Para cada uno de los parámetros libres que se deseen graficar, es necesario que se proporcione un valor Δ (paso de graficación).

Una vez definidos los parámetros del diagrama, el usuario debe de definir los parámetros de la metaheurística que se vaya utilizar (existen valores por defecto). El Algortimo 3 muestra el proceso del trazo de un diagrama de bifurcación con la variación de dos parámetros utilizando metaheurísticas.

Algoritmo 3 Generación de diagrama de bifurcación con dos parámetros ($limInf\_\alpha 1$, $limSup\_\alpha 1$, $limInf\_\alpha 2$, $limSup\_\alpha 2$, $\Delta 1$, $\Delta 2$, $varEdoRangos$, $valParsEstaticos$)

```
1: α1 ← limInf_α1
2: α2 ← limInf_α2
3: inicializar vector_puntos_fijos
4: while α1 < limSup_α1 do
5:    while α2 < limSup_α2 do
6:       agregar metaHeuristica(α1, α2, varEdoRangos, valParsEstaticos) a vector_puntos_fijos
7:       α1 ← α1 + Δ1
8:    end while
9:    α2 ← α2 + Δ2
10: end while
11: graficar vector_puntos_fijos
```

El Algoritmo 3 inicializa los valores de los parámetros libres que se desean graficar con el valor del límite inferior de su rango. Mientras los parámetros no lleguen a su límite superior, se calculan los puntos fijos utilizando alguna metaheurística. Esta metaheurística

es invocada utilizando los rangos de las variables de estado y los valores de los parámetros libres. Esta invocación es un experimento independiente de cualquier otro. La metaheurística tiene como valor de retorno un conjunto (que puede ser vacío) que contiene los puntos fijos encontrados en esa región. Lo que se hace con los puntos fijos contenidos en este conjunto es acumularlos en un vector que contendrá todos los puntos fijos que conformarán el diagrama. Por último, se grafican los puntos contenidos en el vector de puntos fijos para generar el diagrama de bifurcación.

Utilizando el algoritmo de optimización metaheurístico no es necesario tener un punto inicial estable, sólo es necesario proporcionar la región en la que se desea realizar el análisis (los rangos de variables y parámetros). El método busca todos los puntos fijos (no necesariamente estables) del sistema en la región de búsqueda. Además de que el proceso de generar un diagrama de bifurcación con el método propuesto es realizado bajo mínima supervisión; sólo deben ser establecidos los parámetros para el método y éste se encarga de obtener los puntos que conformarán el diagrama y trazarlo.

El proceso para generar un diagrama con la variación de un solo parámetro puede ser intuido del algoritmo 5.2. Las variaciones que tendría, serían la eliminación de las instrucciones relacionadas con el segundo parámetro libre. El procedimiento general es exactamente igual que el mostrado en el algoritmo 5.2.

5.3. JMathPlot

La graficación de diagramas de bifurcación es implementada a través del uso de la librería JMathPlot. JMathPlot [Gobern07]es una herramienta que sirve para dibujar todo tipo de diagramas y permite un alto grado de interactividad con las gráficas. Esta herramienta forma parte de una biblioteca más grande llamada JMathTools, diseñada e implementada en Java2, bajo licencia BSD.

JMathPlot permite crear todo tipo de diagramas, de puntos, de barras, de líneas, de caja y algunos otros. También incorpora una barra con herramientas, algunas sirven para

hacer zoom en determinadas zonas del diagrama, definir los ejes, modificar o definir los datos a representar, etc.

JMathPlot está estructurada como la API de Java2 y como la mayoría de librerías de Java2, en forma de paquetes y sub-paquetes. Toda la biblioteca proviene de *org.math*, en cuyo punto se divide en dos paquetes llamados *io* y *plot*. Cada uno de ellos engloba una funcionalidad que viene determinada por el objetivo de las clases que incluye.

El paquete *io* incluye todas las clases necesarias para la entrada y salida de datos, es decir, clases para cargar o crear archivos, clases para leer o escribir estos archivos (binarios) mediante flujos (paquete *stream*) y clases para interpretar estos archivos (paquete *parser* con su única clase *ArrayString*). También incluye un paquete para controlar el flujo y darle un formato *LittleIndian* (paquete *littleendian*). Además incorpora interfaces para poder imprimir desde el archivo, desde un *String* o del Portapapeles.

El paquete *plot* es el núcleo de la biblioteca y está subdividido en diferentes subpaquetes: *canvas*, *components*, *icons*, *plotObjects*, *plots*, *render* y *utils*. En primer momento nos encontramos las clases *PlotPanel*, sus subclases *Plot2DPanel*, para gráficas en 2D (para diagramas con la variación de un solo parámetro), y Plot3DPanel, para 3D (para diagramas con la variación de dos parámetros). Estos son los paneles de los que hará uso la aplicación, así como de otros paneles auxiliares para cambiar parámetros de la gráfica o ingresar datos como: *DataPanel*, *MatrixTablePanel* o *ParametersPanel*.

El subpaquete canvas incluye la clase abstracta *PlotCanvas* y sus subclases *Plot3DCanvas* y *Plot2DCanvas*. Estas son los paneles sobre los que se dibujarán las gráficas y son el contenedor de todo lo que se dibuja.

El subpaquete *components* incluye las clases del contenido de los paneles: la barra de herramientas del panel de datos, DataToolBar, la barra de herramientas del panel principal, *PlotToolBar*, o el panel de la leyenda, *LegendPanel*, y la propia clase leyenda, *Legend* , incluida dentro *LegendPanel*. También hay clases para representar el panel de

datos cuando se ejecuta como *JFrame*, llamada *DataFrame* y otra para cambiar la escala de la gráfica llamada *SetScalesFrame*.

El subpaquete *plotObjects* contiene los objetos mínimos que pueden ser dibujados y las características que pueden tener estos objetos. Los objetos son los ejes de coordenadas, *Axe*, los nombres, las unidades o en conjunto los caracteres que pueden aparecer, *Label*, las líneas de cualquier tipo, *Line*, o la apariencia de la base de la gráfica y la que contiene las referencias al sistema de coordenadas (*Base*) y los ejes, *BasePlot*. Las características que pueden tomar las clases descritas son un conjunto de interfaces: *Editable*, *Plotable*, *BaseDependant* y *Noteable*. La primera de ellas sirve para editar o cambiar los datos de una gráfica mientras se está ejecutando; *Plotable* define el objeto de una clase como un objeto que se puede dibujar (todos los tipos de gráficas); *BaseDependant* marca las clases cuya definición y / o representación depende de la Base sobre la que está definida (como las clases *Axe* o *BasePlot*). *Noteable* da la posibilidad de destacar el objeto por sobre los demás con un color.

Finalmente el subpaquete *render* contiene las clases "dibujantes", es decir, las clases que trazan las gráficas, los ejes, etc. El paquete viene encabezado por la clase abstracta *AbstractDrawer* seguida de *AWTDrawer* y de sus dos clases *AWTDrawer2D* y *AWTDrawer3D*. También están incluidas una clase abstracta llamada *Projection* y sus clases hijo *Projection2D* y *Projection3D*, utilizadas para coordinarse con los ejes de coordenadas.

5.4. Comentarios finales

En este cápitulo se ha descrito el procedimiento para el trazo de los diagramas de bifurcación. Un diagrama de bifurcación refleja el comportamiento de una de las variables de estado respecto a la variacón de uno o dos parámetros del sistema dinámico. Para construir el diagrama, el usuario requiere proporcionar al sistema: 1) Los rangos de operación de todas las variables de estado que forman parte del sistema dinámico. Estos rangos conforman el espacio de búsqueda del problema de optimización que será resuelto por medio de una metaheurística. 2) El o los rangos en los que se desea se desplacen los parámetros de

bifurcación que formarán parte del diagrama. 3) Un paso de graficación por cada uno de los parámetros que formarán parte del diagrama, los cuales son el incremento que tendrá cada uno de los parámetros para realizar un nuevo experimento utilizando la metaheurística y encontrar los puntos fijos del sistema en esa región. 4) Un valor para los parámetros libres del sistema que no formarán parte del diagrama (parámetros estáticos). 5) Los parámetros propios de la metaheurística seleccionada.

Para construir el diagrama se realizan un conjunto de experimentos independientes con la metaheurística seleccionada por el usuario. Para cada experimento se modifica el valor de alguno de los parámetros de bifurcación, incrementándolo en su paso de graficación realizando una especie de barrido sobre el área definida por el o los rangos de los parámetros de bifurcación. Cada combinación de valores de los parámentros de bifurcación corresponde a una región de operación del sistema dinámico. Cada uno de los experimentos realizados busca los puntos fijos en una región de operación.

Capítulo 6

Resultados

En este Capítulo se presentan los diagramas obtenidos mediante el uso de el sistema desarrollado correspondientes a una serie de sistemas dinámicos. El objetivo de este Capítulo es el de demostrar que la herramienta desarrollada es capaz de: 1) tomar la definición de un sistema dinámico proporcionada por un usuario, validar dicha definición léxica, sintáctica y semánticamente; 2) permitir la selección (por parte del usuario) de algún módulo (que implemente alguna metaheurística) e incorporarla a la herramienta de manera reflexiva al sistema; 3) obtener (bajo mínima supervisión) los puntos fijos en el espacio de búsqueda definido por el usuario: 4) generar diagramas de bifurcación dados los puntos fijos obtenidos. Se considera que cuestiones de interpretación de los diagramas estan ligados a factores como la intención, objetivo, área de estudio, etc. del usuario de la aplicación.

Se muestran los diagramas generados por la aplicación, con la variación de uno y dos parámetros. Los dos primeros casos no tienen ninguna interpretación física y sólo se presentan para ejemplificar el uso del sistema desarrollado. Después se validan las formas normales de algunos tipos de bifurcaciones, presentadas en [Seidel99] y [Kuznetsov98]. Se presenta un caso práctico, el diagrama de bifurcación de una red eléctrica de tres nodos. Por último se comprueba el funcionamiento del mecanismo de adición de módulos, mediante la incorporación de un módulo implementado llamado *Constant*, que extiende la clase *MetaHeuristic*.

6.1. Sistemas de una variable

Considere el sistema dinámico de la Ecuación 6.1

$$\dot{x} = x^2 + a \tag{6.1}$$

Como podemos apreciar es un sistema de una sola variable de estado (x) y un parámetro (a), valor que podemos ajustar. El objetivo es conocer como cambian los puntos de equilibrio del sistema conforme cambiamos a, eso es lo que representa el diagrama de bifurcación. Si se resuelve la ecuación algebraica de la ecuación diferencial mostrada en la Ecuación 6.1, se observa que los puntos fijos x del sistema están dados por $x = \pm\sqrt{-a}$ por lo que sólo existirán puntos de equilibrio para $a \leq 0$. Así, el diagrama fue trazado utlizando rangos que van de -0.5 a 0.5 para la variable x y de -0.25 a 0.25 para a, además de un Δ_a de 0.01.

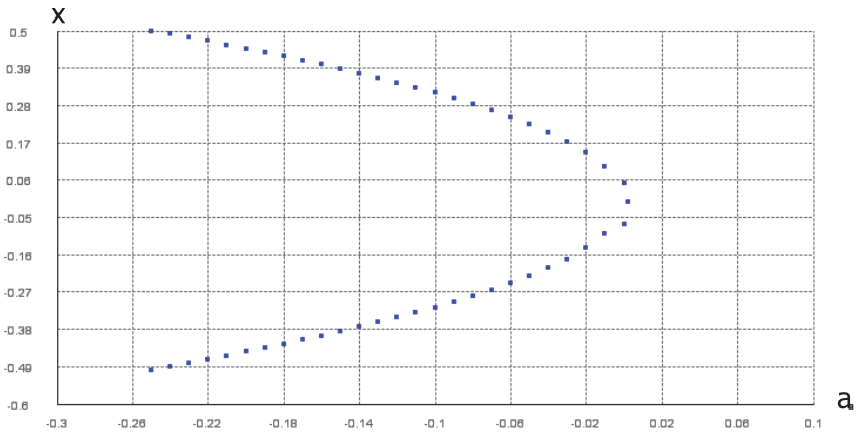


Figura 6.1: Diagrama de bifurcación para el sistema 6.1 con x de -0.5 a 0.5, a de -0.25 a -0.03, Δ_a de 0.02.

Los parámetros utilizados para el algoritmo de enjambre de partículas fueron de 10 iteraciones, 50 partículas, un radio de 0.15, un tamaño de nicho de 5 y los factores de aprendizaje recomendados en [Barrera08] de 2.1. El diagrama de la figura 6.1 es generado en aproximadamente 15 segundos, sin supervisón del usuario. Como se puede apreciar en la Figura 6.1, para $a = 0$ existe un sólo punto de equilibrio en $x = 0$, mientras que para $a < 0$

vemos la aparición de dos puntos de equilibrio en $\pm\sqrt{-a}$, lo cual coincide con el análisis hecho anteriormente.

Ahora considere el sistema definido en el Sistema 6.2

$$\dot{x} = x^2 + a - b \tag{6.2}$$

Es en esencia el mismo sistema planteado en 6.1 con la adición del parámetro b. Por lo que al resolver el sistema algebraico del sistema dinámico representado por la Ecuación 6.2 se observa que los puntos fijos x del sistema están dados por $x = \pm\sqrt{b-a}$ por lo que sólo existirán puntos de equilibrio para $(b-a) \geq 0$. Así, el diagrama fue generado con rangos que van de -0.387 a 0.387 para x, de -0.25 a -0.1 para a y de -0.1 a 0.1 para b que aunque son arbitrarios si podemos deducir que en toda esa región deberan de existir puntos de equilibrio puesto que el menor valor para b será -0.1 y el mayor valor de a es -0.1. Es decir, no existen en esta región valores tal que $(b-a) < 0$. Además se utilizaron un Δ_a de 0.02 y un Δ_b de 0.02. Ver Figura 6.2.

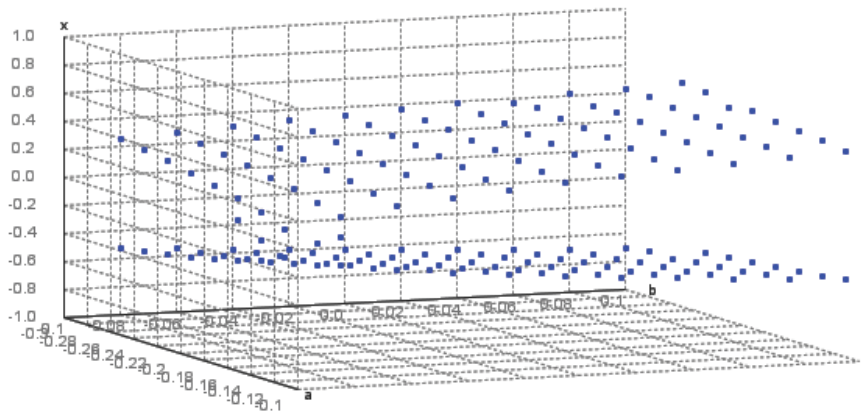


Figura 6.2: Diagrama de bifurcación con la variación de dos parámetros para el sistema 6.2 con x de -0.387 a 0.387, a de -0.25 a -0.1, con b de -0.1 a 0.1, Δ_a de 0.02 y Δ_b de 0.02

En la Figura 6.2 se observa que, en efecto, conforme los parámetros a y b se acercan a -0.1 los puntos fijos colisionan en uno solo ($x = 0$). Y conforme $(b-a) > 0$, comienza la aparición de un par puntos fijos para el sistema ($x = \pm\sqrt{b-a}$).

6.2. Aplicación en formas normales

Un punto fijo asociado a un sistema dinámico, donde la matriz del Jacobiano presenta todos los valores propios con parte real diferente de cero, es llamado un punto fijo hiperbólico. Cuando la condición es violada, las bifurcaciones que presenta el sistema dinámico pueden ser una de las bifurcaciones que surgen en los sistemas representados por las Ecuaciones 6.3, 6.4, 6.5, 6.6.

$$\dot{x} = \mu - x^2 \tag{6.3}$$

$$\dot{x} = (\mu - x)x \tag{6.4}$$

$$\dot{x} = (\mu - x^2)x \tag{6.5}$$

$$\dot{x} = (\mu + x^2)x \tag{6.6}$$

Estas ecuaciones son llamadas formas normales o formas normales topológicas, y son empleadas para reducir un sistema no-lineal dado, a su forma más simple posible, preservando la dinámica en una vecindad del punto fijo donde la condición es violada. En este contexto, un sistema dinámico con un parámetro puede ser reescrito mediante un cambio de variable, como una de las formas normales cerca del punto fijo hiperbólico, teniendo ambos el mismo diagrama de bifurcación. El cambio de variable es regularmente no trivial.

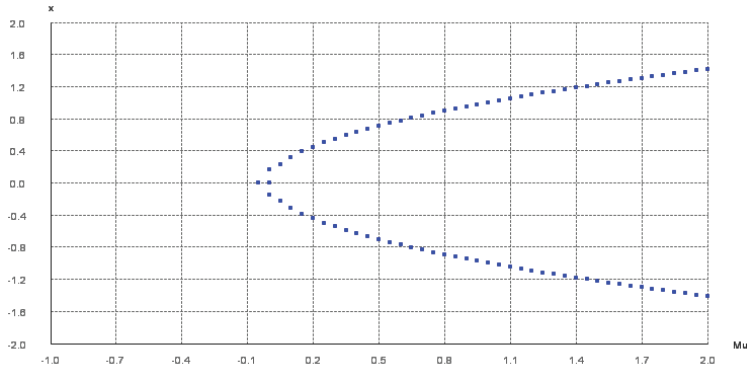


Figura 6.3: Diagrama de bifurcación para la forma normal definida en la Ecuación 6.3

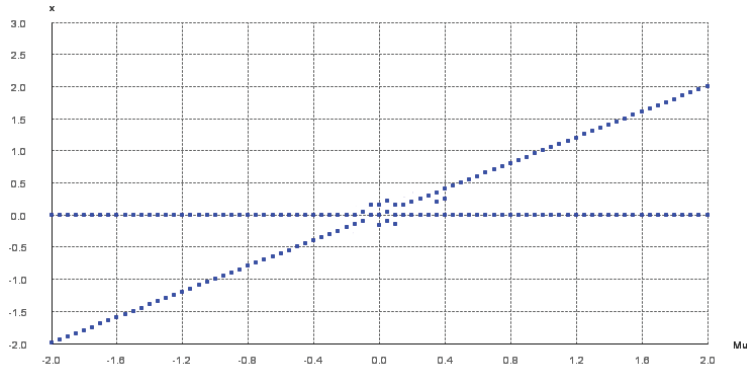


Figura 6.4: Diagrama de bifurcación para la forma normal definida en la Ecuación 6.4

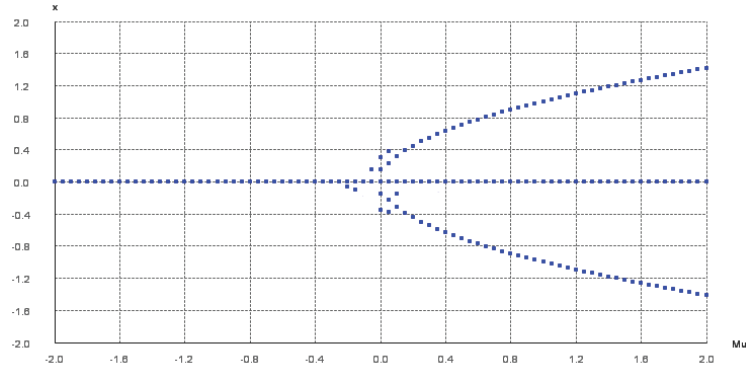


Figura 6.5: Diagrama de bifurcación para la forma normal definida en la Ecuación 6.5

Para las formas normales el sistema fue utilizado para generar los diagramas de bifurcación de cada una. El algoritmo de optimización de enjambre de partículas fue inicializado con un enjambre de 500 partículas, un máximo de 10 partículas por especie, y un valor de radio $r = 0.15$. El espacio de búsqueda para todas las formas normales fue el intervalo [-2, 2], el valor inicial $\mu = -2.0$ se incrementó en $\Delta_\mu = 0.05$ hasta $\mu = 2.0$. Para cada μ_k el algoritmo se ejecutó durante 100 iteraciones. Los diagramas de bifurcación que fueron obtenidos para cada una de las formas normales son mostradas en las Figuras 6.3, 6.4, 6.5, 6.6; La Figura 6.3 representa el diagrama de bifurcación obtenido a partir de la Ecuación 6.3, la cual es la forma normal utilizada para describir la bifurcación *saddle*

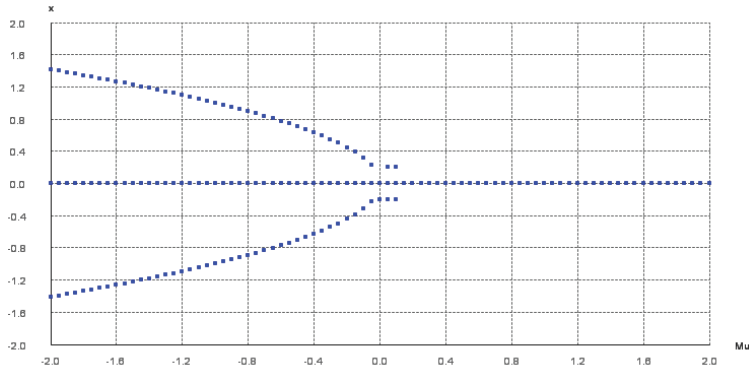


Figura 6.6: Diagrama de bifurcación para la forma normal definida en la Ecuación 6.6

node. Como se puede apreciar cuando $\mu < 0$ no existen puntos de equilibrio en el sistema; cuando $\mu = 0$ aparece un sólo punto fijo; cuando $\mu > 0$ aparecen dos puntos de equilibrio. La Figura 6.4 representa el diagrama de bifurcación obtenido a partir de la Ecuación 6.4 la cual es la forma normal utilizada para describir la bifurcación *transcrítica*. Para cualquier $\mu < 0$ o $\mu > 0$ existen dos puntos fijos, los cuales colisionan en uno solo para $\mu = 0$. Los dos puntos fijos son $x = 0$ y $x = \mu$. La Figura 6.5 representa el diagrama de bifurcación obtenido a partir de la Ecuación 6.5 la cual es la forma normal utilizada para describir la bifurcación *pitchfork supercrítica*. Para los valores negativos de μ existe un punto fijo en $x = 0$. Para $\mu > 0$ hay un un punto fijo en $x = 0$, y dos para $x = \pm\sqrt{\mu}$. La Figura 6.6 representa el diagrama de bifurcación obtenido a partir de la Ecuación 6.6, la cual es la forma normal utilizada para describir la bifurcación *pitchfork subcrítica*. En este caso, para $\mu > 0$ existe un punto fijo en $x = 0$ y hay dos puntos fijos en $x = \pm\sqrt{-\mu}$. Para $\mu > 0$ sigue existiendo el punto fijo en $x = 0$. Estos resultados coinciden con aquellos encontrados en [Seidel99, Kuznetsov98, Barrera08] y son presentados en la Figura 6.7.

Puesto que no se ha complementado el uso de la metaheurística implementada con algún algoritmo númerico que permita dar mayor precisión a la búsqueda de puntos fijos, podemos apreciar la presencia de falsos positivos, es decir, óptimos locales, que en realidad no son puntos fijos del sistema. Esto se debe a que el óptimo se encuentra en una zona plana y/o a que el tamaño del radio seleccionado no es suficientemente grande para

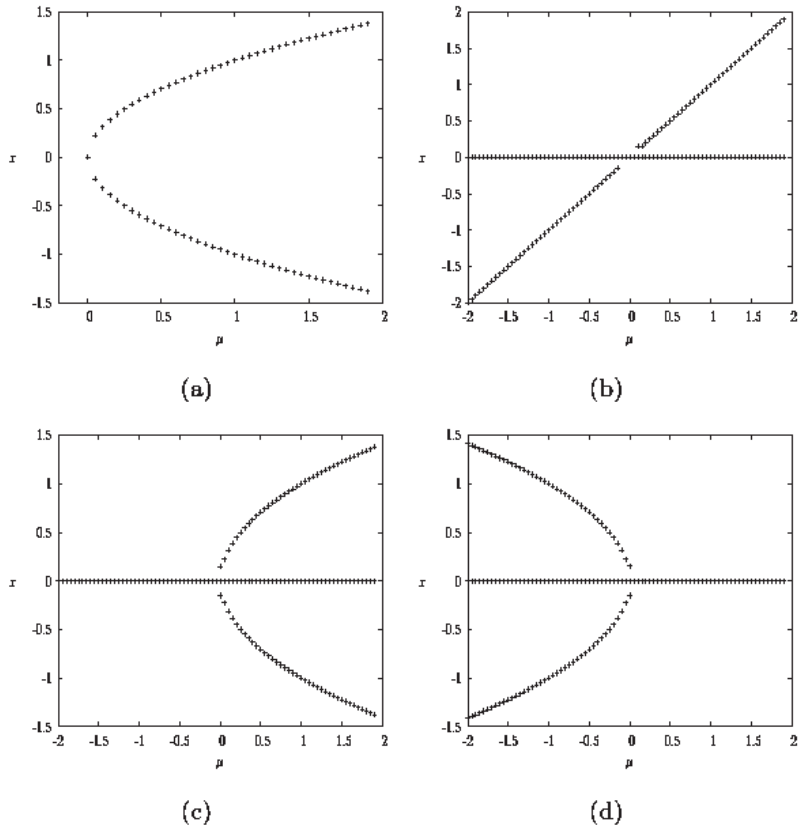


Figura 6.7: Diagramas de bifurcación de las formas normales presentados en [Seidel99, Kuznetsov98, Barrera08]: a) bifurcación saddle node (Ecuación 6.3), b) bifurcación transcrítica (Ecuación 6.4), c) bifurcación pichfork supercrítica (Ecuación 6.5) y 4) bifurcación pichfork subcrítica (Ecuación 6.6).

poder considerar que esos puntos son en realidad el punto fijo. Sin embargo, es posible obtener mejores resultados ajustando el tamaño y radio del nicho, la aptitud de selección y demás parámetros de la metaheurística. En la sección Trabajo Futuro se comenta más al respecto.

6.3. Red eléctrica de tres nodos

La red eléctrica mostrada en la Figura 6.8 ha sido usada en [K.86, Ian Dobson88] para ilustrar el fenómeno de colapso de voltaje que se presenta con la variación de la carga

y en [Barrera08, López10] para demostrar la utilidad de las técnicas metaheurísticas en el trazo de diagramas de bifurcación.

El circuito equivalente de tres nodos debe ser visto como un circuito equivalente a un área local de interés, conectado a una red mayor [Barrera08]. $E_0\angle\theta_0$ es un voltaje constante en magnitud y fase (sin importar el flujo de potencia) y $E_m\angle\delta_m$ es el bus de la terminal del generador. En la Figura 6.8 también se muestran la admitancia compleja de las líneas de transmisión conectadas al generador y al bus infinito, una carga y un capacitor. El voltaje medido en la terminal es $V\angle\delta$.

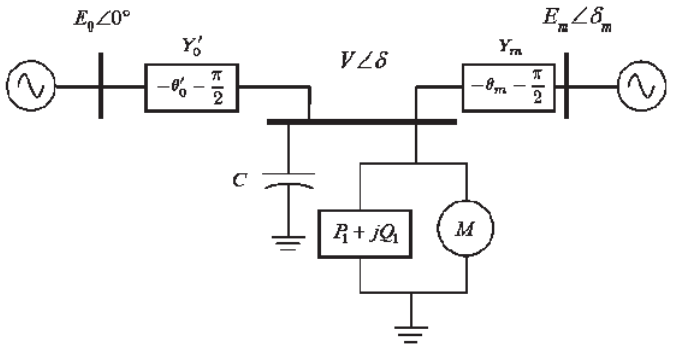


Figura 6.8: Diagrama del sistema eléctrico de tres nodos

La dinámica del sistema está gobernada por las siguientes cuatro ecuaciones diferenciales ordinarias

$$\dot{\delta}_m = \omega \tag{6.7}$$

$$\dot{\omega} = \frac{1}{M}(P_m - D_m\omega + VE_mY_m\sin(\delta - \delta_m - \theta_m) + E_m^2Y_m\sin\theta_m) \tag{6.8}$$

$$\dot{\delta} = \frac{1}{K_{qw}}(-K_{qv2}V^2 - K_{qv}V + Q - Q_0 - Q_1) \tag{6.9}$$

$$\dot{V} = \frac{1}{TK_{qw}K_{pv}}[-K_{qw}(P_0 + P_1 - P) + (K_{pw}K_{qv} - K_{qw}K_{pv})V + K_{pw}(Q_0 + Q_1 + Q) + K_{pw}K_{qv2}V^2] \tag{6.10}$$

En estas ecuaciones las variables de estado están definidas como sigue: δ_m es el ángulo de fase del voltaje del generador, ω es la velocidad del rotor, δ es el ángulo de

fase del voltaje de la carga y V es la magnitud del voltaje de carga. Las funciones P y Q que aparecen en estas ecuaciones representan, respectivamente, la potencia activa y reactiva proporcionadas por la carga de la red. Éstas son dadas por las Ecuaciones 6.11 y 6.12.

$$\begin{aligned} P &= -VE_0'Y_0'\sin(\delta+\theta_0') - VE_mY_m\sin(\delta-\delta_m+\theta_m) \\ &\quad +V^2(Y_0'\sin\theta_0' + Y_m\sin\theta_m) \qquad (6.11) \\ Q &= VE_0'Y_0'\cos(\delta+\theta_0') + VE_mY_m\cos(\delta-\delta_m+\theta_m) \\ &\quad -V^2(Y_0'\cos\theta_0' + Y_m\cos\theta_m) \qquad (6.12) \end{aligned}$$

En las Ecuaciones 6.11 y 6.12, en lugar de incluir el capacitor C en el circuito, un circuito equivalente de Thevenin [K.86] con un capacitor fue derivado con los valores dados por las Ecuaciones 6.13, 6.14 y 6.15.

$$E_0' = \frac{E_0}{\sqrt{1+C^2Y_0^{-2}-2CY_0^{-1}\cos\theta_0}} \qquad (6.13)$$

$$Y_0' = Y_0\sqrt{1+C^2Y_0^{-2}-2CY_0^{-1}\cos\theta_0} \qquad (6.14)$$

$$\theta_0' = -\frac{\pi}{2} + \tan^{-1}\frac{C-Y_0\cos\theta_0}{-Y_0\sin\theta_0} \qquad (6.15)$$

Los otros elementos que aparecen en las ecuaciones son parámetros constantes, relacionados a la carga, la red y el generador. Todos estos parámetros tienen valores fijos durante el análisis, excepto $Q1$, la demanda de potencia reactiva de la carga. Los parámetros de la carga, la red y el generador son dados en la Tabla 6.1

Tabla 6.1: Valores constantes para los parámetros en el sistema eléctrico de tres nodos

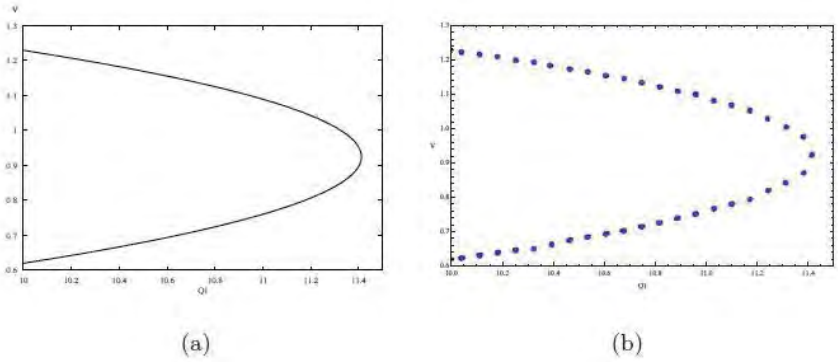
parámetro	valor	parámetro	valor	parámetro	valor
K_{pw}	0.4	K_{pv}	0.3	K_{qw}	-0.03
K_{qv2}	2.1	T	8.5	K_{qv}	-2.8
P_0	0.6	P_1	0	Q_0	1.3
E_0'	2.5	P_m	1.0	E_m	1.0
M	0.3	Y_m	5.0	Y_0'	8.0
θ_0'	-0.2094	Q_1	10.0	θ_m	-0.08726
D_m	0.05				

Para la inicialización del enjambre de partículas se asignaron rangos específicos para cada una de las variables de estado; La Tabla 6.2 muestra estos rangos. El algoritmo de

Tabla 6.2: Valores para los rangos de búsqueda de las variables

variable	rango
δ_m	[0, 1]
ω	[-1, 1]
δ	[0, 1]
V	[0, 2]

optimización de enjambre de partículas fue inicializado con un enjambre de 500 partículas, 20 partículas por especie y un valor de radio $r = 0.1$. Para cada Q_{1k} el algoritmo de optimización de enjambre de partículas fue ejecutado 100 iteraciones. Los puntos fijos que fueron encontrados para la red eléctrica con la herramienta desarrollada y el diagrama de bifurcación generado por el programa AUTO son mostrados en la Figura 6.9.



(a) (b)

Figura 6.9: Diagramas de bifurcación para el sistema eléctrico de tres nodos: a) obtenido con el paquete AUTO y b) obtenido con la herramienta propuesta en esta tesis.

De la Figura 6.9 puede observarse que se logra obtener un diagrama muy parecido al generado por el paquete AUTO. Utilizando el la herramienta propuesta no es necesario tener un punto inicial estable, ni siquiera es necesario un punto inicial, sólo se necesitan los rangos de operación del sistema. El sistema busca todos los puntos fijos (no necesariamente estables) del sistema en la región de búsqueda.

6.4. El módulo *Constant*

Para validar el correcto funcionamiento del mecanismo para la adición de nuevos módulos, se implementó un módulo llamado *Constant*. El cual tiene como única función regresar una constante por cada una de las variables de estado del sistema, sin importar que dicha constante no sea en absoluto un punto fijo del sistema y sin tomar en cuenta el propio sistema dinámico. El propósito es observar la capacidad que tiene el sistema implementado para observar y modificar su estructura de alto nivel, con el fin de utilizar en tiempo de ejecución el código agregado por otros programadores.

Este módulo cumple con el requisito de extender e implementar la clase abstracta *MetaHeuristic*. El sistema requiere, como en cualquier caso, que el usuario proporcione los rangos de las variables de estado y parámetros del sistema, además del paso de graficación de cada parámetro que se encuentre en el diagrama. El único parámetro que requiere la pseudo metaheurística *Constant* es el valor de la constante que devolverá el módulo. Como ya se mencionó, la constante devuelta por el módulo *Constant* es independiente del sistema dinámico y del valor de sus variables.

Para probar el módulo *Constant* se utilizó el sistema representado por la Ecuación 6.16.

$$\begin{aligned} \dot{x} &= y + a \\ \dot{y} &= \frac{y}{b} \end{aligned} \tag{6.16}$$

Se trazó el diagrama de bifurcación para la variable de estado x para la cual se paso al módulo *Constant* un valor constante de 2, un rango de [-1.5 a 1.5] para el parámetro a y un rango de [2 a 4] para el parámetro b. La Figura 6.10 muestra el plano generado por el sistema. Aunque los puntos y el sistema son irrelevantes en este caso, el sistema fue capaz de interpretar los metadatos generados por la máquina virtual de Java en el archivo *Constant.class* e incorporarlos a su propia estrucura como un nuevo módulo. Este

mecanismo es el que permite que diversos programadores desarrollen sus propios módulos y los incorporen al sistema. El código necesario para implementar este módulo se encuentra en el Apéndice A.

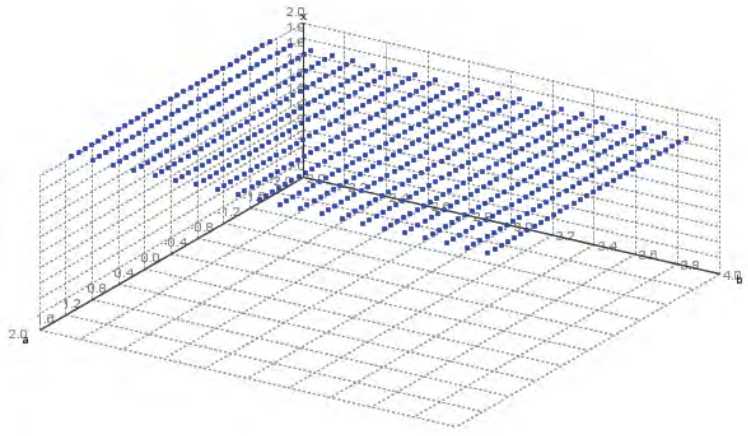


Figura 6.10: Diagrama generado por el módulo *Constant* para el Sistema 6.16

6.5. Comentarios finales

A través del sistema desarrollado es posible definir un sistema dinámico y obtener cualquier número de diagramas de bifurcación, experimentando con distintos rangos y combinaciones de variables y parámetros. Permite construir diagramas con la variación de uno o dos parámetros. Para la generación de los diagramas es posible seleccionar cualquier módulo disponible que implemente alguna metaheurística (para la búsqueda de puntos fijos). Una vez generado un diagrama, el sistema permite (entre otras cosas) el cambio de la escala del diagrama, el movimiento de los ejes, seleccionar puntos específicos de la gráfica, realizar acercamientos o alejamientos a regiones específicas del diagrama, convertir el diagrama a imagen en formato *PNG*, etc.

Hasta aquí se han mostrado algunos de los resultados obtenido mediante el uso del sistema. Cada uno de los experimentos fue ejecutado mediante mínima supervisión y sin el conocimento prévio del comportamiento del sistema. Si bien la precisión de los resultados

puede ser mejorada, estos resultados pueden ser tomados como punto de inicio para otros métodos.

Capítulo 7

Conclusiones y trabajo Futuro

En este último Capítulo se resumen los resultados obtenidos, se presentan las conclusiones a partir de los mismos, además se presentan las adecuaciones y trabajo futuro que complementarían el presente trabajo.

7.1. Conclusiones Generales

En la presente tesis se ha mostrado la estructura general de la implementación de un sistema para el trazo de diagramas de bifurcación a través del uso de metaheurísticas. El sistema tiene un nicho de aplicación tanto en personas interesadas en el área de teoría de bifurcaciones (usuarios finales), como en personas interesadas en el área de optimización a través de metaheurísticas (usuarios programadores), ya que permite la adición de módulos desarrollados de manera independiente a la aplicación.

La implementación se puede dividir en cuatro fases. La primera corresponde a la implementación de un compilador que reconoce sistemas de ecuaciones diferenciales de primer orden y que genera un *script* en código Java que puede se ejecutado de manera dinámica y que representa la función objetivo en la búsqueda de puntos fijos del sistema. El compilador desarrollado sólo puede reconocer sistemas de ecuaciones diferenciales de primer orden, si se deseara ingresar sistemas de mayor orden, es responsabilidad del usuario simplficar el sistema a uno de primer orden equivalente. Este compilador permite que el

usuario final del sitema introduzca modelos de sistemas dinámicos a través de una interfaz gráfica, sin necesidad de modifcar ni recompilar el código fuente de ningún programa, como ocurre con [Barrera08] y [López10].

La segunda fase corresponde al mecanismo que permite ejecutar módulos para la búsqueda de puntos fijos que fueron desarrollados de manera independiente a la aplicación principal. Lo cual se logra a través del uso del API *Java reflection*, de la extensión de la clase *MetaHeuristic* y de la importación de la clase *SystemFunction*. Lo que permite a los usuarios avanzados (programadores) introducir módulos de metaheurística con un mínimo de esfuerzo. Es responsabilidad del programador la validez de su técnica. La herramienta sólo gráfica los puntos de equilibrio proporcionados por la metaheurística.

La tercera parte está relacionada con la implementación de una técnica metaheurística que será utilizada por defecto para la búsqueda de puntos fijos del sistema. Fue elegida la optimización de enjambre de partículas propuesta en [Barrera08]. El algoritmo de optimización de enjambre de partículas con nichos funciona de manera adecuada en problemas multimodales ya que tiende a encontrar todos los óptimos de la función, sin embargo depende de parámetros adicionales; uno de ellos, el radio, es de particular interés. El valor que sea asignado a este parámetro no sólo afecta el tiempo de ejecución del algoritmo, sino que también puede afectar la correcta localización de los óptimos de una función. El problema de encontrar un valor adecuado para el radio, aún es un problema abierto de algoritmos evolutivos.

La cuarta parte del trabajo, corresponde a la graficación de los puntos fijos del sistema, es decir al trazo del diagrama de bifurcación. Mediante el uso de *JMathPlot* fue posible implementar una serie de herramientas que son de utilidad en el anális del diagrama y permiten una generación y manipulación intuitiva de los diagramas (a nivel de interfaz).

Sin embargo no se han implementado las estructuras y métodos necesarios que permitan analizar la estabilidad de los puntos de equilibrio del sistema, con lo que se pierde

la posibilidad de detectar algunos tipos de bifurcaciones, de esto se comenta en la sección dedicada al trabajo futuro.

Aunque la precisión del algoritmo implementado no supera a la de los algoritmos numéricos estándar, sí proporciona una herramienta de búsqueda automatizada. Esta sola característica implica un ahorro de tiempo considerable para un investigador; aún sin tener mucha información de la función que se analiza, es posible encontrar buenas aproximaciones, algunas veces en cuestión de minutos. Aún si la búsqueda de puntos fijos tardara días, se realiza de forma completamente automatizada, sin necesitar la supervisión del investigador. Si los puntos obtenidos no tienen una precisión adecuada, pueden ser usados como puntos iniciales para los algoritmos numéricos estándar. Además de que no depende de las propiedades de la función a analizar como son la continuidad y la diferenciabilidad.

Por último, el sistema desarrollado contiente una serie de herramientas que permiten almacenar y administrar los modelos y diagramas generados. El sistema brinda a los usuarios: generación diagramas con distintos parámetros para un mismo modelo, realizar acercamientos y manipular el diagrama desde distintos ángulos (en el caso de los diagramas en tres dimensiones), conocer la posición de cada punto fijo del diagrama mediante el uso del cursor del *mouse*, cambiar algunas de las leyendas y colores del diagrama, cambiar la escala de los ejes, guardar el conjunto de los puntos que conforman el diagrama en un archivo ASCII, guardar el diagrama como una imagen en formato *png*, almacenamiento permanente, etc.

Por lo tanto esta tesis tiene utilidad tanto en el área de algoritmos evolutivos, como en el estudio de sistemas dinámicos aplicado a cualquier área. Proporciona un marco de trabajo que elimina parte de la implementación a programadores que deseen desarrollar técnicas metaheurísitcas para la búsqueda de puntos de equilibrio y auxilia en el análisis de sus resultados. El sistema permite a investigadores interesados en el análisis de sistemas dinámicos, trazar diagramas de bifurcación con las ventajas que provee el uso de técnicas metaheurísticas. El sistema proporciona una serie de herramientas visuales y administrativas que lo hacen una buena alternativa como herramienta auxiliar.

7.2. Trabajo Futuro

La herramienta que aquí se describió, aún puede ser mejorada en gran medida. En cuanto al reconocedor y compilador, la gramática puede ser extendida para reconocer diferenciales de orden superior y para poder reconocer elementos discretos en el sistema (como modelar un interruptor). Además se puede implementar una recuperación de errores más eficiente que el modo *pánico*, que de una descripción más precisa de los errores. Es de gran importancia generar la forma matricial del sistema durante la generación de código, para que ésta pueda ser utilizada en temas de estabilidad y mejora de la precisión de los diagramas.

El proceso de ejecución de los módulos de búsqueda de puntos fijos puede ser mejorado al implementar un mecanismo de validación de los puntos de equilibrio del sistema, de tal manera que se garantice que los puntos devueltos por la técnica metaheurística sean en realidad puntos fijos (y no falsos positivos). Además es posible implemetar técnicas numéricas (gradiente por ejemplo) que aumenten la precisión de las técnicas metaheurísticas, haciendo uso de la matriz del sistema.

El sistema desarrollado no contempla el cálculo de la estabilidad de los puntos fijos del sistema, cuestión en la que es posible seguir trabajando con el fin de lograr una herramienta más robusta y de mayor utilidad. Para lo cual sería necesario (además de la construcción del sistema en su forma matricial y de un mecanismo que permita sustituir los valores del punto fijo en dicha matriz) implementar un método de diferenciación numérica, que permita el cálculo del Jacobiano de la matriz del sistema y un mecanismo para la diagonalización de dicho Jacobiano con el propósito de obtener los valores propios de éste, y así determinar la estabilidad de cada uno de los puntos fijos. Alternativamente es posible implementar un método que no haga uso de métodos numéricos sino que de manera manual perturbe un Δ cada uno de los puntos fijos en cada una de las dimensiones del mismo, con el propósito de conocer más detalles de la estabilidad de cada uno de los puntos fijos y más importante aún, con el fin de que éste sea el mecanismo utilizado en caso de encontrarse

funciones no diferenciables o con discontinuidades.

A nivel de interfaz, tambien es posible realizar mejoras. Es posible utilizar componentes visuales que permitan observar más de un diagrama simultáneamente. Un panel que contenga información acerca de un punto de equilibrio seleccionado en el diagrama como: su posición, su estabilidad, etc. Modificar la manera de trazar los diagramas con el fin de poder mostrar con distintos colores puntos de equilibrio con distintas características de estabilidad. Para ello sería necesario agregar la información de la estabilidad a cada uno de los puntos fijos y de modificar el método para la graficación, de tal manera que cada uno de los puntos fijos sea un objeto distinto dentro de la misma y no que el conjunto de los puntos de equilibrio (el vector de puntos fijos) sea el único objeto a graficarse, como hasta ahora.

De esta manera el presente trabajo de tesis puede ser usado como base para el desarrollo de un sistema de mayores alcances y más robusto de tal manera que sea una herramienta para el análisis de sistemas dinámicos que presenten características que dificulten su análisis con los métodos numéricos estándar, como puede ocurrir en sistemas con discontinuidades o con regiones donde las funciones no sean diferenciables, así como sistemas híbridos modelados con funciones tanto continuas como discretas. Siendo útil en cualquier área de la ciencia interesada en teoría de bifurcación.

Apéndice A

Código del módulo *Constant*

```
//Librerias necesarias para la implementacion
import Inter.MetaHeuristic; //Clase abstracta que se debe extender e implementar
import Inter.SystemFunction; //Clase que se utiliza para evaluar el sistema dinamico

public class Constant extends MetaHeuristic{
   //Variable de la constante
   double constant;

   public Constant(SystemFunction of, double r[][])
   {
      //El constructor de la clase padre (MetaHeuristic)

      super(of,r);

      /* se inicializa la constante con el
       * valor limite inferior del rango
       * del primer parametro de bifurcacion
       */
      constant = r[0][0];
   }

   //El metodo que se invoca para el calculo de los puntos fijos
   //Y es aqui donde se debe implementar el codigo de la metaheuristica

   public double [][] findFixedPoint(double [] freeParams, double [][] ranges)
                                                                  throws Exception
   {
      //Se construye el vector de estado con el valor de la constante
      //para todas las variables de estado

      double [][]stateVector=new double[1][this.ranges.length];
      //ranges.length es la longitud del vector de estado (numero de variables)

      for(int i=0;i<this.ranges.length;i++)
         stateVector[0][i]=constant;
      return stateVector; // se devuelve el vector de estado
   }

   //Este metodo es invocado por el sistema para obtener los nombres
   //de los parametros de la metaheuristica y sus valores por defecto

   public String [][] getMetaParamsName()
```

```
   {
      String [][]metaParams= new String [1][2];
      //Nombre
      metaParams[0][0]="Constant value";
      //Valor por defecto
      metaParams[0][1]=Double.toString(constants);

      return metaParams;
   }

   //Este metodo es invocado por el sistema para obtener
   //la cadena que tiene la descripcion de la metaheuristica
   //El programador es el responsable de darle formato a esta cadena

   public String getMetaHuristicDescription()
   {
      String description = "This is NOT a module to calculate fixed points.\n" +
          "This is just a test of the module appending mechanism.\n" +
          "You should set a constant value. This value \n" +
          "is returned by this module to construct a diagram."
          +" The values must be in a double value\n " +
          "otherwise the module will take the lower \n" +
          "bound value of the first parameter range.";

      return description;
   }

   //Este metodo es invocado por el usuario para asignar
   //los valores seleccionados por el usuario para la metaheuristica
   // es responsabilidad del programador validad que los parametros que
   //provienen del sistema sean validos. El arreglo s contiene los parametros,

   public void setMetaParams(String [] s)
   {
       //se valida que el parametro sea una variable "double"
        if(isDouble(s[0]))
           //se convierte la cadena a double y se asigna al parametro constant
           constant=Double.parseDouble(s[i]);
   }

  //funcion para validar si una cadena contiene un numero flotante
   private static boolean isDouble(String s)
   {
      try
      {
         Double.parseDouble(s);
         return true;
      }
      catch(NumberFormatException nfe)
      {
         return false;
      }
   }
}
```

Referencias

[Aggarwal00] Aggarwal, V. Solving transcendental equations using genetic algorithms, 2000.
URL `http://web.mit.edu/varun ag/www/techwrite.html`

[Barrera08] Barrera, J. *Análisis de sistemas dinámicos utilizando herramientas de inteligencia artificial*. Tesis Doctoral, Universidad Michoacana de San Nicolás de Hidalgo, 2008.

[Clerc02] Clerc, J., M. y Kennedy. the particle swarm: explosion, stability,and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[Cota03] Cota, E. J. G. *Guia práctica de ANTLR*, 2003.
URL `www.lsi.us.es/ troyano/documentos/guia.pdf`

[Crina Grosan06] Crina Grosan, A., A. A. y Gelbukh. Evolutionary method for nonlinear systems of equations. *Advances in Artificial Intelligence*, 2:283–293, 2006.

[Doedel09] Doedel, E. J. *AUTO-07P : Continuation and bifurcation software for ordinary differential equations*, 2009.
URL `http://indy.cs.concordia.ca/auto`

[Gobern07] Gobern, A. A. Diseño e implementación de didácticas para facilitar el aprendizaje de principios de la física. Inf. téc., Universidad de Lleida Escuela Politécnica Superior, Lleida España, Septiembre 2007.

[Govaerst00] Govaerst, W. *Numerical methods for bifurcations of dynamic equilibria.* Engineering and Computer Science. Society for industrial and applied mathematics, 1ª ed$^{\text{ón}}$., 2000.

[Haupt04] Haupt, S. E. H., Randy L. *Practical Genetic Algorithms.* John Wiley & Sons, New Jersey, 2004.

[Ian Dobson88] Ian Dobson, H. D. C., J. S. T. A model of voltage collapse in electric power systems. *En Conference on Decision and Control*. 1988.

[J. Timmis08] J. Timmis, N. O., P. Andrews. An interdisciplinary perspective on artificial immune systems. *Evolutionary intelligece*, 2008.

[K.86] K., W. Modeling of power systems components at severe disturbances. *En International conference on large high voltage electric systems*. 1986.

[Kennedy J.01] Kennedy J., E. R. C. y. S. Y. *Swarm intelligence.* Morgan Kaufmann, San Francisco, 2001.

[Kuznetsov98] Kuznetsov, Y. A. *Elements of applied bifurcation theory.* Springer-Verlag, New York, 1998.

[Li X.06] Li X., B. J. y. B. T. Particle swarm with speciation and adaptation in a dynamic environment., 2006.
URL `http://doi.acm.org/10.1145/1143997.1144005`

[López10] López, R. *Diagramas de bifurcación de funciones discontinuas o no diferenciables.* Proyecto Fin de Carrera, Universidad Michoacana de San Nicolás de Hidalgo, 2010.

[Mahfoud95] Mahfoud, S. W. *Niching methods for genetic algorithms.* Tesis Doctoral, Urbana, IL, USA, 1995.

[Parr93] Parr, T. J. *Language Translation Using PCCTS and C++. A Reference Guide.* Automata Publishing Company, 1072 South DeAnza Blvd, San Jose, California 95129 USA, 1993. ISBN 0-9627488-5-4.

[Regis99] Regis, O. R. G. *Software para en análisis visual interactivo de sistemas dinámicos discretos en la circunferencia*. Proyecto Fin de Carrera, Universidad Autónoma del Estado de México, 1999.

[Sacks91] Sacks, E. P. Automatic analysis of one - parameter planar ordinary differential equations by intelligent numeric smulation. *Artificial Intellince*, 48:27–56, 1991.

[Scowen98] Scowen, R. S. Extended bnf, a generic base standard. Inf. téc., University of Cambridge, Great Britain, September 1998.
URL `http://www.cl.cam.ac.uk/ mgk25/iso-14977-paper.pdf`

[Seidel99] Seidel, R. *World of bifurcation. online collection and tutorials of nonlinear phenomena*, 1999.
URL `http://www.bifurcation.de/`

[Sommers06] Sommers, F. Terence parr introduces antlr 3.0. *Leading-Edge Java*, 2006.
URL `http://www.artima.com/lejava/articles/antlr_3.html`

[Storn Rainer95] Storn Rainer, P. K. Differential evolution - a simple and efficient adaptative scheme for global optimization over continuous spaces. Inf. Téc. TR-95-12, Berkely, California, March 1995.

[Valbuena08] Valbuena, S. J. *Programación Avanzada en Java*. Elizcom, 2008. ISBN 9584446010.