ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Curso Académico 2010/2011

Proyecto de Fin de Carrera

SIMULACIÓN MEDIANTE APPLETS DE JAVA:
DINÁMICA Y CONTROL DEL OSCILADOR DE DUFFING

Autor: Jorge Plaza Pulido

Tutores: Inés Pérez Mariño y Jesús M. Seoane Sepúlveda

Agradecimientos

Me he querido guardar este rincón para dar las gracias a todas aquellas personas que han estado apoyándome a lo largo de estos años.

Gracias a mis compañeros de clase. Muchos compañeros de clase y de prácticas pasan a lo largo de la carrera, pero algunos de ellos ya no son sólo compañeros, ya son grandes amigos. En cierto momento me dijeron que esta es una de las mejores etapas de la vida y es cierto, por eso me llevo grandes personas de todo este tiempo.

Gracias a mis tutores del proyecto. Gracias a los profesores Inés Pérez y a Jesús M. Seoane del Departamento de Física de la U.R.J.C., sin los cuales no estaría terminando ahora mis estudios universitarios y que me han ayudado muchísimo durante el desarrollo de este proyecto, dedicándome gran parte de su tiempo inculcándome sus conocimientos sobre los sistemas caóticos en general y sobre el Oscilador de Duffing en particular.

Gracias a otros muchos profesores que han puesto su granito de arena para que yo acabe siendo Ingeniero Técnico en Informática de Gestión.

Y sobre todo, gracias a mi familia y amigos que me han apoyado en los buenos y en los malos momentos durante estos últimos años.

Resumen

El proyecto que se ha desarrollado y que se explica a continuación consiste en el diseño e implementación de una aplicación informática que permite simular el funcionamiento de un sistema físico, concretamente un sistema dinámico no lineal y así observar su comportamiento caótico a lo largo del tiempo.

En nuestro caso, el sistema físico sobre el cual se basa el proyecto es el llamado *Oscilador de Duffing*. El ingeniero eléctrico alemán Georg Duffing (1864-1944), fue la primera persona que se encargó del estudio de dicho sistema a principios del siglo XX, cuando estudiaba el movimiento de pandeo de una viga al colocarse encima de ella un peso *P.* Desde ese momento se conoce como Oscilador de Duffing.

Respecto a la parte informática, se ha decidido desarrollar esta aplicación mediante la programación de un *applet* de Java por las ventajas que se detallan a continuación. Este lenguaje de programación se encuentra encuadrado dentro del paradigma de programación orientado a objetos, en este caso incluido en el entorno de desarrollo *NetBeans*. Desde el principio se decidió que la aplicación se programaría como *Applet de Java*, puesto que su uso en la actualidad está muy extendido y a su vez permite su utilización a través de Internet y desde cualquier punto que esté conectado a la red. Son utilizables desde cualquier tipo de navegador web como cualquier otra web. Los más usuales en estos tiempos son *Google Chrome, Mozilla Firefox, Internet Explorer* o *Safari*, entre otros. Además el lenguaje de programación Java cuenta con una propiedad importante que es la portabilidad. La portabilidad permite que una aplicación desarrollada en este lenguaje pueda ser utilizada en cualquier sistema operativo.

La aplicación en un principio está enfocada al mundo de la física, ya sea a nivel docente como científico. Gracias a su diseño agradable y su fácil utilización puede utilizarse incluso sin tener nociones sobre el fundamento físico de este sistema caótico.

Índice

1. Introducción

El proyecto que se ha realizado simula el comportamiento de un sistema físico. Los sistemas físicos pueden clasificarse según diferentes criterios. En nuestro caso, nuestra aplicación simulará el funcionamiento del *Oscilador de Duffing*.

Un **sistema dinámico** es un sistema que sufre cambios a lo largo del tiempo que vienen definidos generalmente en términos de ecuaciones diferenciales (sistemas continuos) ó ecuaciones en diferencias (sistemas discretos). En nuestro caso concreto estudiamos un sistema continuo que viene modelizado por una ecuación diferencial ordinaria de segundo orden.

A su vez, estos sistemas dinámicos pueden ser lineales o no lineales. La diferencia radica en la respuesta del sistema a la presencia de estímulos externos. Un sistema lineal responderá siempre de forma proporcional al estímulo externo, mientras que los sistemas no lineales son muchos más difíciles de analizar, ya que su comportamiento ante dicho estímulo no guarda relación lineal con la causa que lo produce. El *sistema de Duffing* es un **sistema no lineal**. En términos físico-matemáticos, los sistemas no lineales son aquellos que quedan definidos por ecuaciones no lineales y que necesitan métodos matemáticos que utilizan la integración de las ecuaciones que definen el comportamiento de dicho sistema. Ejemplo de estos métodos son el método de *Euler*, el método del punto medio o el método de *Runge-Kutta*.

Los sistemas no lineales pueden presentar comportamiento caótico (impredecible). Se entiende que un **sistema es caótico** cuando es sensible a las condiciones iniciales, es decir, que pequeños cambios en ellas, producen grandes diferencias de comportamiento del sistema en el futuro. En este sentido, puede decirse que los sistemas caóticos son deterministas, ya que se conoce de forma precisa la secuencia que les da origen, es decir, la ley que rige su evolución, pero, sin embargo, son impredecibles para periodos largos de tiempo, debido a su alta sensibilidad a las condiciones iniciales. La diferencia entre valores que inicialmente están muy próximos unos de otros, crece exponencialmente con el paso del tiempo, produciendo así un comportamiento en el sistema imposible de conocer de antemano.

Edward Lorenz (Figura 1), matemático y meteorólogo, fue el primer científico que se propuso estudiar el comportamiento caótico de los sistemas físicos. Su motivación surgió mientras estudiaba un modelo de predicción meteorológica [1]. Pronto se dio cuenta que en estos sistemas, variaciones en

las condiciones iniciales producían grandes cambios en el estado final del sistema.

Figura 1.
Edward Lorenz

Lorenz dio a conocer el famoso "Efecto Mariposa" (Figura 2), mediante el cual podía explicar sus teorías sobre los sistemas caóticos. Este término se ha utilizado para describir la divergencia exponencial que sufren los sistemas tras pequeñas variaciones en las condiciones iniciales. Esta es una de las características fundamentales de este tipo de sistemas y del nuestro en particular. Las variables varían de forma compleja, lo que hace imposible la predicción del comportamiento del sistema a partir de un determinado momento.

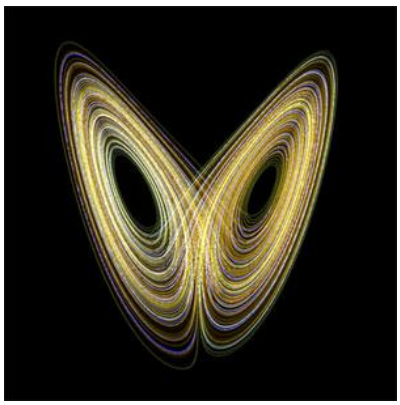


Figura 2.
Atractor de Lorenz y el Efecto Mariposa

Actualmente existe un gran interés en el mundo de científico y de la ingeniería sobre este campo. Gracias a las nuevas tecnologías, cada vez es más fácil explicar numéricamente el comportamiento de dichos sistemas, que abundan en la naturaleza.

En nuestro caso, dentro de este tipo de sistemas, nos centraremos en el *Oscilador Caótico de Duffing* [2]. Éste fue diseñado por el ingeniero eléctrico alemán, Georg Duffing (1864-1944) (Figura 3) a principios del siglo XX, con el fin de estudiar el movimiento de pandeo de una viga [3][4].

Figura 3.
Georg Duffing

Particularmente, el *Oscilador de Duffing* es un sistema dinámico y no lineal, que puede presentar un comportamiento caótico en función de los valores que tomen sus variables. En el caso de que este oscilador sea forzado periódicamente, puede presentar un comportamiento caótico. Cuando dicho sistema sufre oscilaciones pequeñas, su comportamiento puede ser lineal, mientras que a medida que van siendo mayores, el caos va apareciendo en el sistema.

Uno de los aspectos fundamentales a la hora de realizar el estudio de un sistema físico es hacer una análisis geométrico del mismo. Cuando se analiza geométricamente un sistema, hay que tener en cuenta lo que se conoce como **espacio de fases**. Se trata de una construcción matemática que permite representar el conjunto de posiciones y velocidades de un sistema de partículas. Si un sistema describe un movimiento periódico, al completar un ciclo totalmente, volvería a la posición inicial (Figura 4), mientras que si el movimiento fuese caótico, el espacio de fases iría acumulando puntos, de forma que se iría llenando de puntos después de un largo periodo de tiempo (Figura 5).

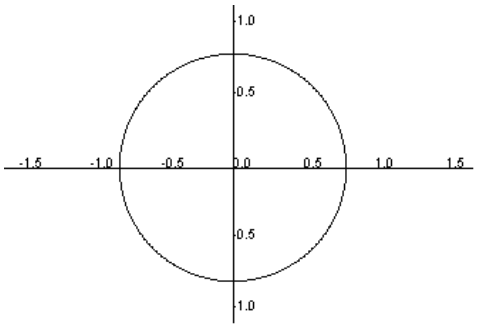


Figura 4.
Movimiento Periódico

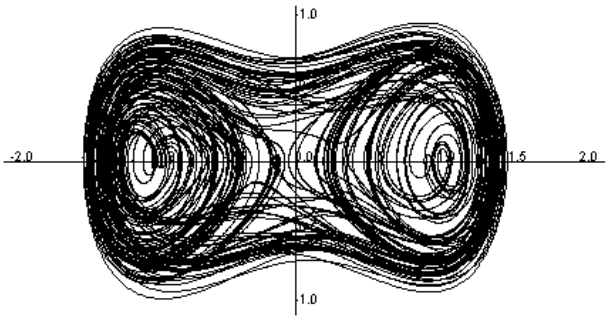


Figura 5.
Movimiento Caótico

2. Objetivos y Metodología

La finalidad de este proyecto es la creación de una aplicación informática que simule el comportamiento de un sistema caótico no lineal, en este caso el *Sistema de Duffing*. Se podrá visualizar la evolución de dicho sistema a lo largo del tiempo a partir de unas condiciones iniciales y una serie de parámetros, que serán definidos por el propio usuario.

La funcionalidad de nuestro *applet*, permitirá mostrar el movimiento de una o dos partículas al mismo tiempo, o bien el movimiento de pandeo que sufrirá una viga cuando se coloca sobre ella una masa (*m*), que ejerce sobre ella una fuerza, $F = m \cdot g$. El usuario podrá elegir a través de la interfaz qué gráfico quiere ver representado. Además, se pretende implementar alguna técnica que nos permita controlar el movimiento caótico de nuestro sistema.

2.1 Descripción del Problema

La ecuación de movimiento del Oscilador de Duffing para una partícula de masa unidad viene representada por la siguiente expresión matemática:

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta x^3 = F cos\omega t$$

donde, δ es el parámetro de disipación, α y β son los coeficientes que determinan el tipo de pozo potencial, ω es la frecuencia del control aplicado y **F** es la amplitud del forzamiento externo.

Un aspecto interesante además en los sistemas caóticos es el control del caos. Todo sistema dinámico no lineal puede ser controlado mediante **técnicas de control** [5]. Estas técnicas se basan en la estabilización del sistema mediante pequeñas perturbaciones que ayudan a regular su comportamiento. Estas perturbaciones no tienen que ser muy grandes para evitar cambios importantes en el comportamiento del sistema inicial. En este caso se ha utilizado la conocida como "Técnica de Control de la Fase" [6], en la cual se introduce una pequeña perturbación en uno de los parámetros, quedando la ecuación que rige nuestro sistema como se muestra a continuación:

$$\ddot{x} + \delta\dot{x} + \alpha x + \beta(1 + \varepsilon cos(r\omega t + \phi))\ x^3 = F cos\omega t$$

y en la que el término de control viene descrito mediante los siguientes parámetros: ε es la amplitud del control aplicado, **r** es la relación entre las

frecuencias aplicada del control ($r\omega$) y la del forzamientos externo (ω)y ϕ que es el parámetro más importante en el control, cuyo significado es la diferencia de fase entre la perturbación aplicada y el forzamiento externo.

En ambos casos, el sistema presentará un comportamiento periódico o caótico en función de las condiciones iniciales y el valor de los parámetros de las ecuaciones que se hayan elegido. Estos comportamientos podrán ser controlados mediante nuestra técnica de control, en la que la diferencia de fase ϕ jugará un papel fundamental en la misma.

2.2 Estudio de las alternativas

2.2.1 El método matemático

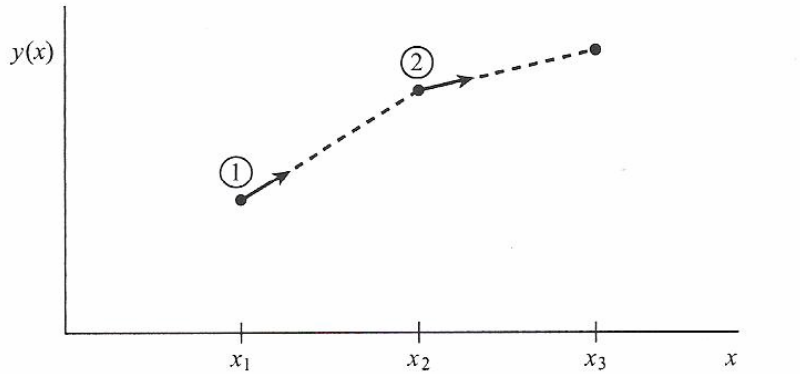
Como se ha explicado anteriormente, los sistemas dinámicos no lineales no pueden resolverse sin utilizar métodos numéricos [7] que integren las ecuaciones que describen dicho sistema y que nos ayudan a aproximarnos a una solución real. Estos métodos son los que se utilizan para la resolución de ecuaciones del siguiente tipo:

$$\frac{dy}{dx} = f\,(x, y)$$

Existen numerosos métodos numéricos de integración. Aquí explicaremos el método más simple y otro más robusto, que será el que utilizaremos en este proyecto.

Método de Euler

En nuestro caso, tratamos de utilizar dos métodos. Primero recurrimos al Método de *Euler* que no es demasiado complejo, pero en el caso de nuestro problema, encontrar una solución utilizando esta metodología se nos plantea complicado por su escasa exactitud.



Este método se basa de forma general en la pendiente estimada de la función que es objeto de estudio, para extrapolar desde un valor anterior a un valor nuevo siguiente. Con la siguiente expresión se puede entender mejor:

Nuevo valor = Valor anterior + Pendiente x Tamaño de Paso

Formalmente, la ecuación que describe el Método de *Euler* es la siguiente:

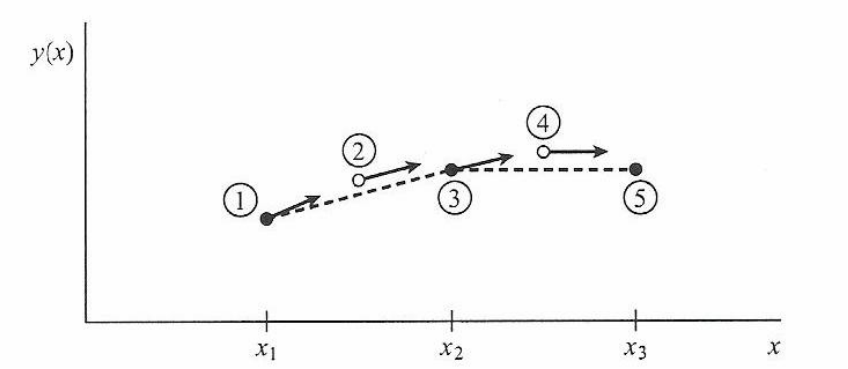
$y_{n+1} = y_n + hf(x_n,y_n) + O(h^2)$

En esta ecuación, h es lo que se conoce como paso de integración y *O* es el error asociado, que en este caso es de orden de h^2.

Tras su estudio, queremos indicar que no es el método más adecuado para el tipo de problema que vamos a abordar (de acuerdo a la experiencia), pues, ya que no se obtiene una buena aproximación a la solución esperada debido a que el error asociado va aumentando a la vez que lo va haciendo *h* y tiene una exactitud de primer orden.

Método del Punto Medio

Este método no es sino una mejora del método de *Euler*. Cuenta con una exactitud de segundo orden, mayor que la del método explicado anteriormente y obtenido mediante la derivada inicial de cada paso con el fin de encontrar el punto medio en cada uno de los intervalos. En la siguiente gráfica, los puntos rellenos indican los valores reales de la función, y los huecos aquellos que se descartan tras el cálculo y uso de sus derivadas.



Método de Runge-Kutta

Es el método que utilizamos para dar solución a nuestro problema. Al igual que los anteriores, es un método genérico enfocado a la resolución de ecuaciones diferenciales y que cuenta con una exactitud muy alta. Esta metodología data de principios del siglo XX y fue desarrollada por los matemáticos Carl Runge y Martin Wilhelm Kutta (Figura 6).

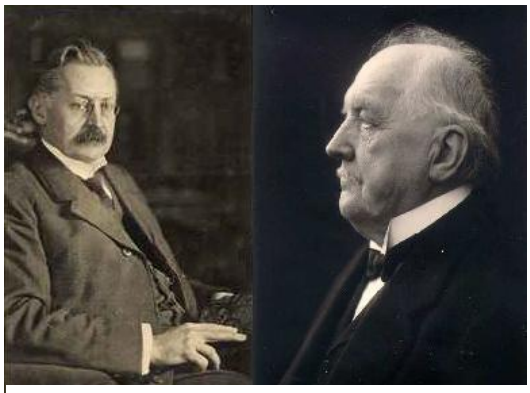
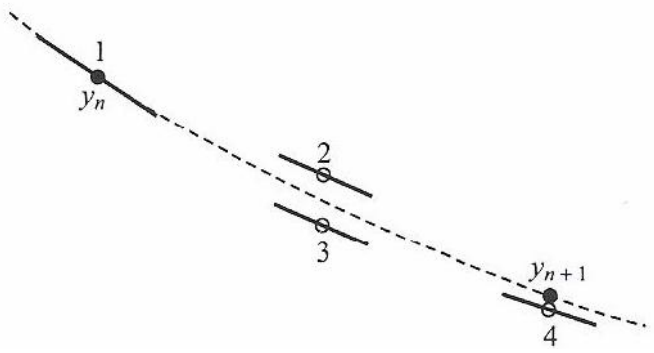


Figura 6.
C. Runge y M.W. Kutta

Realizar los cálculos manualmente es muy complicado, pero cuenta con la ventaja de que su programación para ordenadores es sencilla. Podríamos decir que a su vez se divide en cuatro submétodos. Se elige una anchura de paso *h*, y se calculan cuatro valores. El primero de ellos encontraría un punto inicial (y_n,1), posteriormente se llegaría a dos puntos medios (2,3) y luego al que podría ser un posible punto final (4). Después de esto, y a partir de estas derivadas se calcula el valor de la función a la que querríamos llegar (y_{n+1}).



Según este procedimiento, a partir de un valor inicial de *y* en el instante *x*, se obtiene un valor de y en un instante *x+h*. Los valores intermedios calculados se corresponderían en las siguientes ecuaciones con *k1, k2, k3 y k4*.

$$k_1 = hf(x_n, y_n)$$

$$k_2 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\right)$$

$$k_3 = hf\left(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\right)$$

$$k_4 = hf(x_n + h, y_n + k_3)$$

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

El método de *Runge-Kutta* de cuarto orden tiene un error de paso de orden de h^5, siendo su error total acumulado del orden de h^4. Así pues, el error es mínimo y por ello nos hemos decantado por la utilización de este método. En nuestro caso, teniendo en cuenta que hemos utilizado una anchura de paso de *h=0.01,* el error que estaremos manejando será del orden de $1/10^{10}$, reduciéndolo considerablemente frente al que tendríamos en cualquiera de los métodos anteriores.

2.2.2 Alternativas en el diseño

A lo largo del tiempo de programación de esta aplicación han ido surgiendo diferentes alternativas para el diseño de la interfaz. Desde un primer momento, uno de los objetivos que se marcaron fue la sencillez de la aplicación. Sin desviarnos de este objetivo primordial, durante el desarrollo, se ha ido modificando nuestra interfaz (Figura 7) a medida que se iban incorporando

nuevas funcionalidades. La facilidad de uso de cualquier aplicación es uno de los factores más importantes y a tener en cuenta durante el desarrollo.

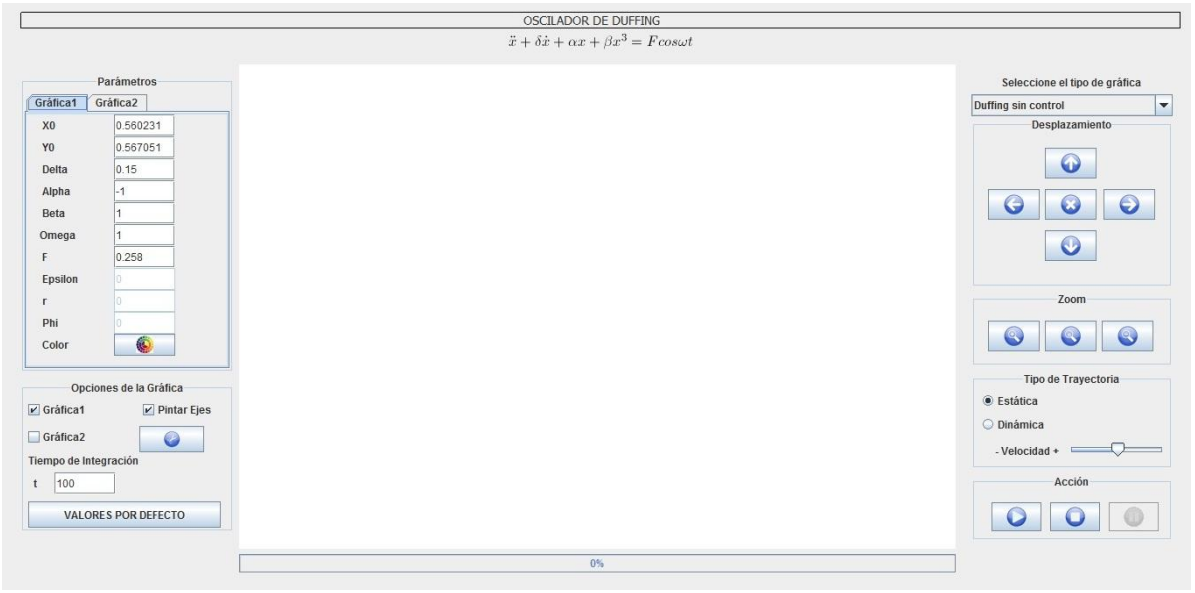


Figura 7.
Visión general del *Applet*

2.3 Metodología de Desarrollo

Para explicar la metodología que hemos utilizado, hemos de entender primero qué es la **Ingeniería del Software** [8]. Ésta se puede entender como la disciplina de ingeniería que se encarga de todos los aspectos relacionados con la producción de software desde sus etapas más tempranas de la especificación del sistema hasta el mantenimiento del sistema tras su puesta en marcha. Mediante el uso adecuado de teorías, herramientas y métodos se tratará de dar solución a los problemas que surgen durante todo el proceso de desarrollo.

A raíz de la explicación anterior, surge el concepto de **metodología de desarrollo de software**. Se trata de un conjunto de procedimientos, técnicas y ayudas a la documentación para el desarrollo de productos software. Las técnicas indican cómo debe ser realizada una actividad técnica determinada identificada en la metodología. Existen diferentes metodologías de desarrollo o modelos de proceso: modelo en cascada, incremental, basado en construcción de prototipos y en espiral. En este caso nos hemos decantado por el modelo incremental (Figura 8) que explicamos a continuación.

El modelo incremental

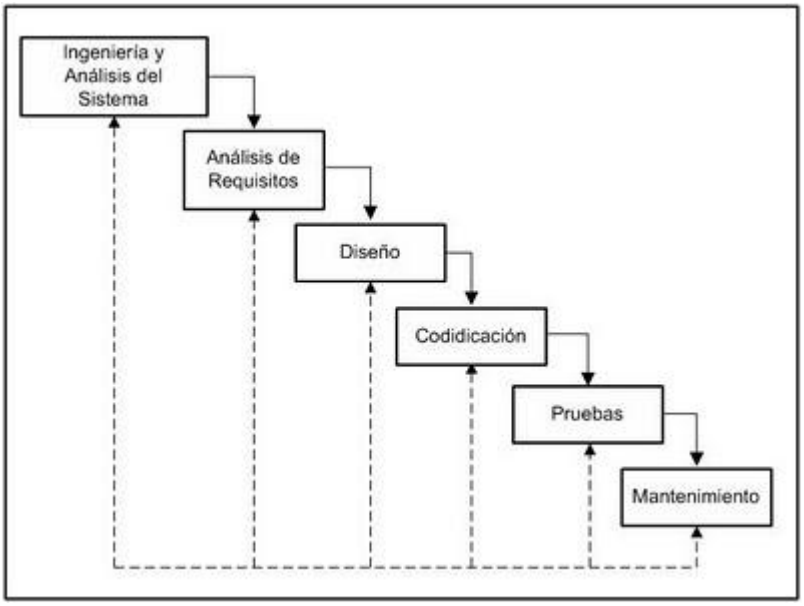


Figura 8.
Esquema del Modelo Incremental

Se trata de un modelo no lineal que comprende diferentes etapas que se conocen como versiones (Figura 9). Cada versión implica mejoras en la versión anterior, a la cual se le han añadido nuevas funcionalidades. Al final de cada etapa se produce la integración de resultados.

VERSIÓN ANTERIOR + NUEVA FUNCIONALIDAD	=	NUEVA VERSIÓN

Lo importante de esta metodología es que con el paso del tiempo y según evoluciona el desarrollo se va creando un software más completo, desde una primera versión simplificada hasta una versión final perfectamente ajustada a las peticiones del usuario. Las primeras versiones sirven al usuario como prototipo de cómo se está desarrollando y de que los requisitos críticos están muy probados.

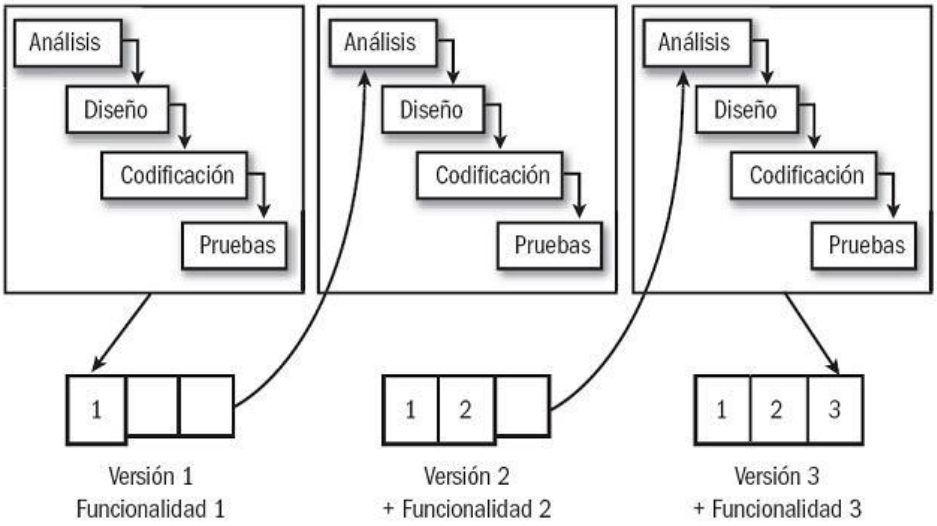


Figura 9.
Evolución de un Sw. según el modelo incremental

Para este proyecto, hemos tenido que seguir cada una de las etapas que se reflejan en los diagramas anteriores. La figura anterior es extensible para *n* versiones.

- **Análisis.** Es una de las tareas más complejas. En ella se marcan los objetivos que hay que alcanzar. Para ello fue necesario conocer el problema del *Sistema de Duffing* y así buscar el método correcto que nos permita resolverlo. Además, se plantean una serie de requisitos relacionados con la funcionalidad y el aspecto de nuestro software. En cada una de las etapas se irán incluyendo nuevos requisitos que completarán los inicialmente exigidos por el usuario.

- **Diseño y codificación**. Partiendo de la base de que *Java* es un lenguaje de programación encuadrado dentro del paradigma de la Programación Orientada a Objetos (POO), en esta etapa se diseñan las clases que utilizaremos. Las clases principales serán diseñadas al inicio, aunque se podrán ir añadiendo métodos y atributos en ellas según sea necesario, o incluso otras clases que nos ayuden en la implementación. Cualquier elemento añadido no restará al conjunto, sino que siempre sumará funcionalidad a nuestra aplicación.

- **Pruebas**. Al final de cada una de las etapas, se realizarán las pruebas necesarias para comprobar que el nuevo incremento, integrado en la versión anterior, haga que la funcionalidad haya mejorado y que el sistema funcione correctamente.

- **Mantenimiento**. Tras todo lo anterior, se evalúan los logros conseguidos y se comprueba si es necesario añadir más funcionalidad. En este caso, volveríamos a la primera etapa, y en caso contrario la creación del producto habría finalizado.

3. Descripción informática

Como ya hemos comentado anteriormente, para el desarrollo de la aplicación que se nos ha pedido, hemos creído que la mejor opción era implementarla como un *applet* de Java. Por ello es necesaria una breve explicación sobre los *applets*.

Un *applet* es un programa escrito en lenguaje Java que puede ser incluido dentro de una página HTML, de igual forma que puede incluirse por ejemplo una imagen. El navegador web lo ejecuta dentro del espacio de la página, lo cual hace que pueda ser utilizado por cualquier usuario que tenga un explorador web y la máquina virtual de Java (JVM). También puede ejecutarse con el *AppletViewer* de *Sun*.

Podemos destacar que los *applets* tienen la ventaja de poder ejecutarse en cualquier sistema operativo para el cual exista alguna versión de la máquina virtual de Java. Como desventaja, los navegadores web necesitan el *plugin* de Java para su ejecución, y éste no está disponible para todos.

3.1 Especificación

Cuando se va a desarrollar un producto software, una de las primeras etapas es el análisis y captura de requisitos. Consiste en definir una serie de objetivos que ha de cumplir nuestra aplicación. Se diferencia entre los requisitos funcionales y no funcionales.

REQUISITOS FUNCIONALES

- El usuario podrá introducir las condiciones iniciales y parámetros de la ecuación del *Oscilador de Duffing*.
- El programa deberá representar la evolución temporal del movimiento definido por el *Sistema de Duffing,* tanto para una órbita como para dos.
- El programa mostrará el pandeo de una viga de acuerdo al *Sistema de Duffing.*
- Para los dos puntos anteriores existe tanto la opción controlada como la no controlada.
- En el caso de la representación temporal que muestra las órbitas descritas a lo largo del tiempo, se podrá elegir entre una representación estática o

dinámica, en la cual se permitirá elegir la velocidad con la que se dibuja la órbita.

- Además, en el caso anterior, tanto para la representación estática como dinámica, existirán herramientas de zoom y desplazamiento sobre los ejes para mejorar la visión de la gráfica.

REQUISITOS NO FUNCIONALES

- La aplicación deberá implementarse como *applet* de Java.
- La interfaz deberá ser agradable y de fácil utilización para el usuario.
- La interfaz será ajustable en función de la resolución del monitor del ordenador que la ejecute.
- La velocidad de representación será aceptable y los tiempos de espera serán cortos.
- Su funcionamiento será sencillo, lo cual permitirá un aprendizaje fácil para usuarios inexpertos.

3.2 Diseño

Como ya ha sido explicado, el proyecto consiste principalmente en diseñar una interfaz gráfica que programaríamos en Java. Este lenguaje de programación dispone de diferentes librerías gráficas con las cuales podemos trabajar. En este caso hemos utilizado el *paquete Awt* y sobre todo el *paquete Swing* que explicamos un poco más detalladamente a continuación.

Swing no es sino una ampliación del paquete *Awt.* Gracias a las numerosas clases con las que cuenta, facilita mucho el trabajo de diseño gráfico. Para el diseño de esta interfaz hemos utilizado diferentes componentes. Cada componente cuenta con métodos propios para trabajar con ellos, incluidos en este paquete que se exponen a continuación:

- ***JPanel***: encuadrado dentro de la familia de los elementos conocidos como contenedores. Se utiliza para organizar y agrupar otros componentes. En nuestro caso se han utilizado varios *JPanel* que nos ayudan en la organización de nuestra interfaz.

- ***JTabbedPane***: se trata de un componente que nos permite distribuir otros elementos en diferentes pestañas. Basta con pinchar sobre la pestaña que queramos utilizar para poder hacer uso de los elementos que estén encuadrados dentro de esa "caja". En este caso se ha utilizado para distinguir las condiciones iniciales y parámetros de las gráficas 1 y 2.

- ***JLabel***: este componente nos ayuda a colocar textos y/o imágenes que no podrán ser modificados por el usuario, solamente por el programador. Los textos que aparecen en la aplicación al empezar a ejecutar, son todos elementos *JLabel*.

- ***JButton***: como en el caso anterior, puede contener tanto textos como imágenes, pero con una diferencia fundamental. A partir de una serie de eventos sobre este botón, pueden desencadenarse diferentes acciones. El evento más utilizado a lo largo de nuestro proyecto es "hacer *click*" sobre los botones. Ejemplo de ellos son la selección de color de la gráfica, botones de zoom, de desplazamiento o de acción.

- ***JCheckBox***: se trata de un elemento que se suele utilizar principalmente a la hora de seleccionar distintos elementos que no se excluyen entre sí. Puede tomar valores de *True* o *False*. Ejemplos de utilización en nuestra aplicación serían las opciones de pintar la gráfica1, la gráfica2 y/o los ejes.

- ***JRadioButton***: muy similar al elemento anterior, pero con la particularidad de que en este caso la selección de una de las opciones excluye a las demás. Para ello es necesario utilizarlo junto a otro elemento llamado ***ButtonGroup***. Un ejemplo de ellos es la selección de pintar las gráficas de forma estática o dinámica.

- ***JComboBox***: se trata de un componente que permite mediante una lista desplegable seleccionar una de las distintas opciones. Aquí se ha utilizado para elegir el tipo de gráfico que se va a representar.

- ***JTextField***: este componente permite introducir o editar textos en tiempo de ejecución. En nuestro caso, se ha utilizado para introducir las condiciones iniciales y parámetros que definen la ecuación de *Duffing*.

- ***JSlider*:** este elemento permite seleccionar gráficamente un valor dentro de un rango específico. Para nosotros, nos ha sido útil para elegir la velocidad con la que se dibujan las gráficas.

- ***JProgressBar*:** mediante este componente se muestra el progreso de la representación que se esté ejecutando.

A continuación, se explicarán los componentes que se han integrado entre sí para el diseño general del *applet* que hemos desarrollado.

- **Encabezados del *applet***: se tratan de dos *Label* independientes en los cuales se muestra el título del *applet* y la ecuación del movimiento que se está utilizando en ese momento, teniendo en cuenta que varía en función de si el sistema está controlado o no.

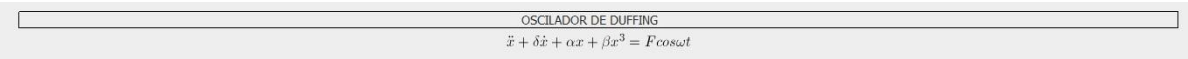


Figura 10.
Cabecera de la aplicación y ecuación del sistema

- **Panel de parámetros:** se trata de un panel constituido por un *JTabbedPane* que a su vez consta de dos *JPanel*, uno para la órbita1 y otro para la órbita2. Estos dos paneles son exactamente iguales y están formados por una serie de *Labels* y de *JTextField* en los cuales se introducen tanto los valores de las condiciones iniciales como de los parámetros de la ecuación. Además, también existe un *JButton*, cuya pulsación permitirá la elección del color de la órbita en cuestión.

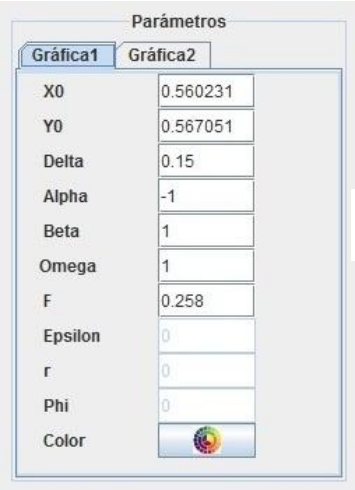
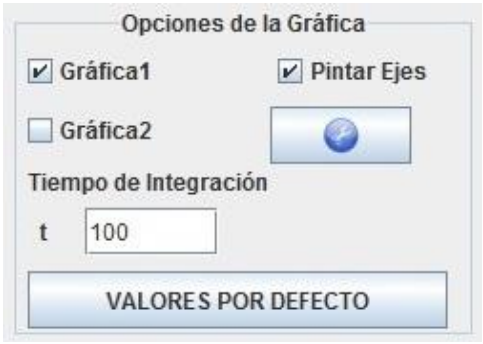


Figura 11.
Panel de Parámetros y Condiciones Iniciales

- **Panel de opciones de la gráfica:** dentro de este panel nos encontramos con tres *jCheckBox*. Las dos primeras nos permiten seleccionar qué órbitas queremos representar (órbita1, órbita2 o las dos) y la tercera nos ofrece la opción de mostrar los ejes *x* e *y* junto a la representación de las gráficas. Además, en este panel encontramos el *jButton* que hemos denominado *Cortar*. Cuando la representación se hace de forma dinámica, cada vez que se pulsa dicho botón, los puntos representados anteriormente se borran, pero la

representación continúa con el resto de puntos. Por otro lado, tenemos un *jLabel* y otro *jTextField*, que nos permite introducir el valor del Tiempo de Integración que consideremos oportuno. Por debajo, la pulsación del botón *"VALORES POR DEFECTO"* hace que parámetros y condiciones iniciales del panel "Parámetros" tomen los valores de inicio de la ejecución.

Figura 12.
Panel de Opciones de la Gráfica



- En la parte derecha del *applet* podemos encontrar un *JComboBox* que nos permite elegir el **tipo de gráfica** que queremos representar. Existen cuatro opciones: *Duffing* sin control, *Duffing* Controlado, pandeo de una viga sin control y pandeo de una viga controlado.

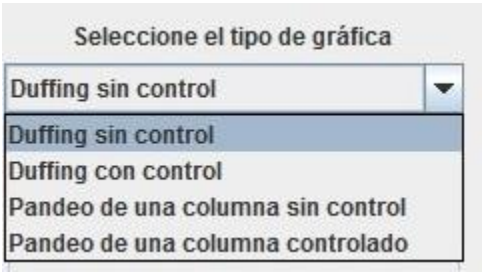


Figura 13.
Panel de selección del tipo de gráfica

- **Paneles de Desplazamiento y de Zoom:** ambos paneles están compuestos por varios *jButton*. El primero de ellos consta de 5 botones con los cuales el usuario puede desplazarse dentro del área de representación (arriba, abajo, izquierda, derecha y volver a posición inicial). En cuanto al zoom, la pulsación de sus botones permite alejar o acercar el gráfico que se ha representado para así conseguir una visión óptima de la representación. También restaurar las dimensiones iniciales. Ambos paneles permanecen activos para las gráficas *Duffing* sin control y Duffing controlado, ya que en el caso de los pandeos no tendría ningún sentido útil su utilización.

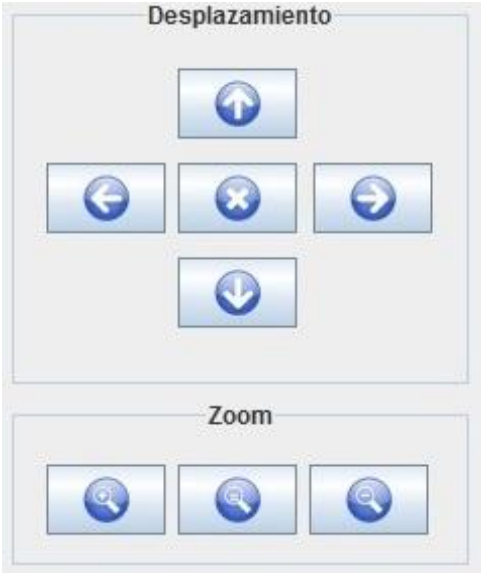


Figura 14.
Paneles de desplazamiento y zoom

- **Panel Tipo de Trayectoria:** este panel está compuesto de dos *JRadioButton*, relacionados por una clase del tipo *RadioButtonGroup*, cuyo cometido es la selección de una única de opción. En este caso nos permite elegir entre una representación estática o dinámica. Para el caso de la representación dinámica, además mediante un *JSlider* se puede variar la velocidad con la que se representa la gráfica. Este panel también tiene sentido en el caso de la representación de las órbitas, puesto que para el pandeo de la viga, solo se permite la representación dinámica cuya velocidad viene definida por los valores resultantes de las ecuaciones.

Figura 15.
Panel de Selección del tipo de Trayectoria

- **Panel de Acción:** solamente está compuesto de tres botones:
  - *Play*: mediante su pulsación comienza la representación del gráfico que haya sido seleccionado por el usuario. Además habilita el botón de pause.
  - *Stop*: cuando se pulsa este botón, se borra la última representación, dejando todo listo para volver a iniciar la ejecución.
  - *Pause*: inicialmente deshabilitado. Mediante su pulsación se detiene, hasta que se vuelva a "*clickar",* la representación dinámica de cualquiera de las gráficas.

Figura 16.
Botones de Acción

- **Panel de Representación y Barra de Progreso:** se trata de un panel en el cual se muestra la representación de la gráfica que haya sido seleccionada por el usuario. Bajo dicho panel se puede ver un *JProgressBar*, que muestra el progreso de la representación superior.

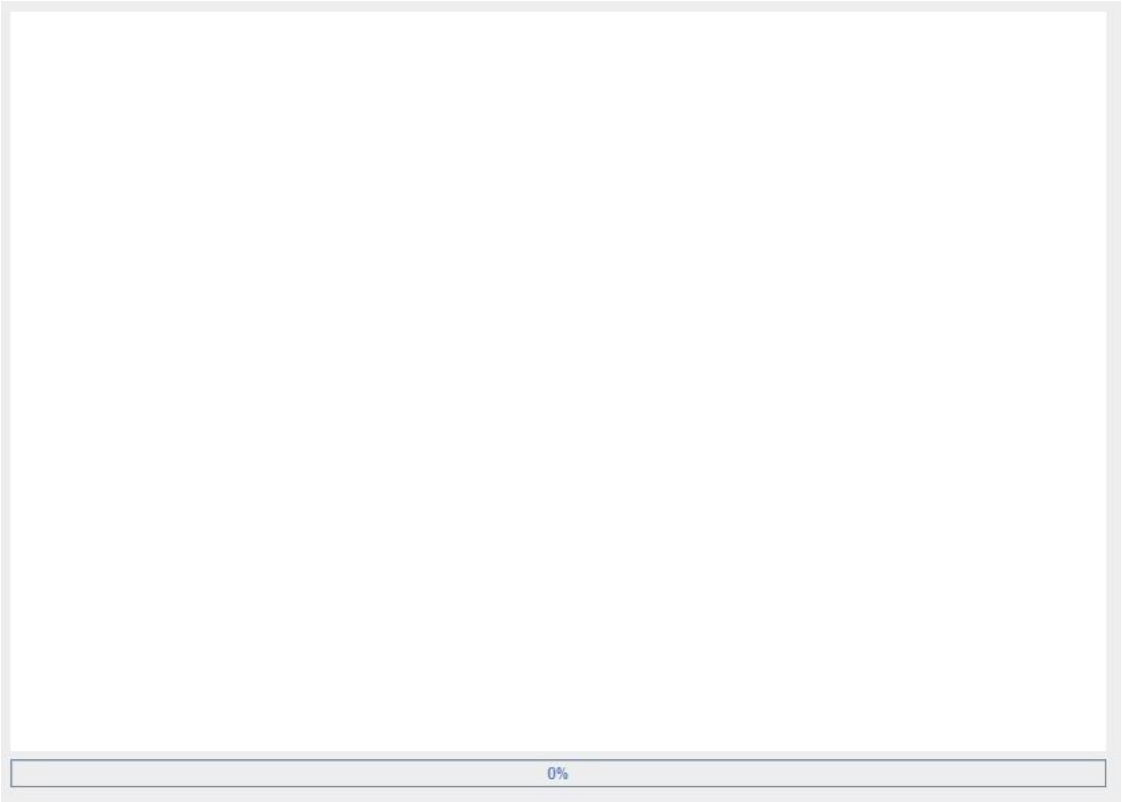


Figura 17.
Panel de Representación

- **El *applet* en conjunto**: la interfaz de nuestro *applet* por tanto es el conjunto resultante de la unión de todos los componentes que hemos explicado anteriormente y cuya diseño final es el que se muestra a continuación:

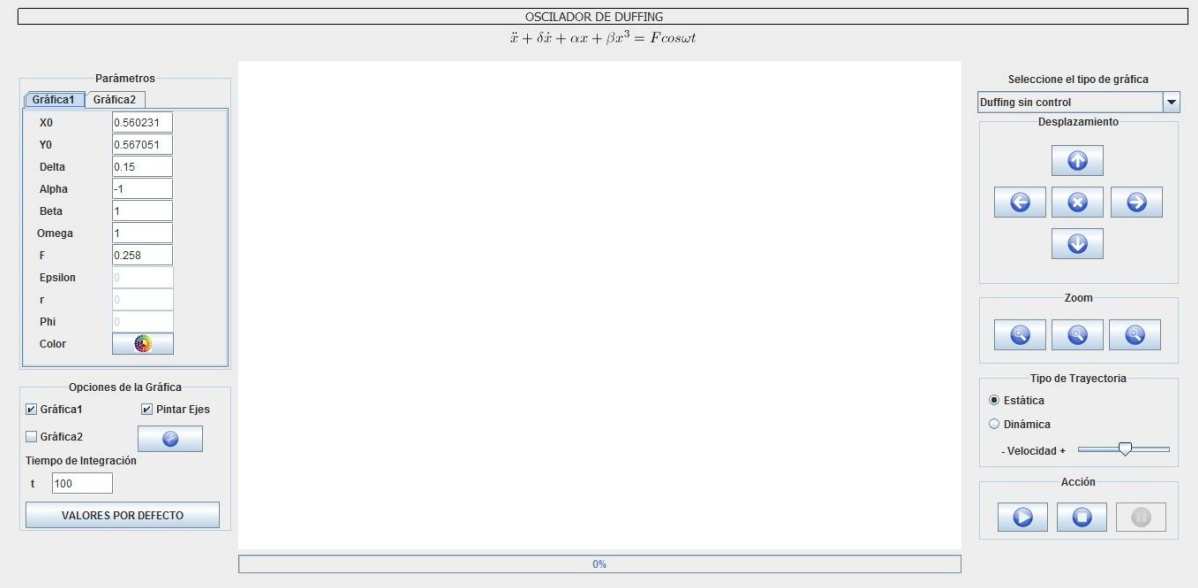


Figura 18.
Vista de la interfaz de la aplicación

- **Diseño de clases**

Una vez recopilada la información necesaria para comenzar a desarrollar nuestro producto, hemos de plantearnos qué clases formarán parte de nuestro proyecto. Al principio se creía que se necesitaban una serie de clases, pero a medida que el proyecto ha ido avanzando, se han definido otras que nos han sido de gran ayuda de cara a la implementación de nuestra aplicación:

- **Clase AppletOsciladorDuffing**

Esta clase hereda de la superior *JApplet*, pudiendo así utilizar cualquiera de los métodos propios de un *applet.* Es la clase que se lanza con la ejecución de nuestra aplicación y nuestra clase principal. El resto de clases se conectan a ella mediante una relación de agregación de composición, es decir, que si eliminásemos esta clase, el resto no tendrían ningún sentido por sí solas. En esta clase se encuentra la definición de la interfaz del *applet*, es decir, todos los elementos visuales y objetos con los que el usuario puede interactuar con el sistema. El método *init()* es uno de los principales de esta clase y se encarga de organizar el applet y posteriormente mostrarlo tal como se diseñó con anterioridad.

En esta aplicación hay que destacar por su importancia los *Threads.* Son segmentos de código que se ejecutan de forma secuencial y paralela a otras secciones de código del programa. La Máquina Virtual de Java es un sistema *multi-Thread*, lo que significa que puede ejecutar varios *hilos* al mismo tiempo. En nuestra aplicación, el uso de *threads* es utilizado en las representaciones gráficas, ya que nos permite dibujar varias órbitas a la vez y hacerlo a velocidades adecuadas. Además, los threads cuentan con una serie de métodos

interesantes que, por ejemplo, nos permiten pausar el proceso de dibujado de una órbita. También han sido utilizados en la representación del pandeo de la viga.

- **Clase Punto**

Es una clase muy sencilla que nos permite crear puntos con sus coordenadas *x* e *y*. Por tanto, un objeto de clase punto solamente cuenta con dos atributos (*x,y*) y los métodos de acceso a ellos (*gets* y *sets*). Además, tenemos un método que nos permite dibujar un punto a través de sus coordenadas. Es de gran utilidad puesto que gráficamente, lo que se representa son secuencias de puntos y con ello se nos facilita mucho el trabajo.

- **Clase Trayectoria**

Esta clase tiene como cometido aglutinar en sus atributos las condiciones iniciales, parámetros de la ecuación, tiempo de integración y tipo de gráfico que ha elegido el usuario para que sea representado. Los únicos métodos existentes dentro de esta clase son los que permiten el acceso a los atributos (*gets* y *sets*).

- **Clase RungeKutta**

Es la clase que se encarga de realizar todos los cálculos matemáticos. Como su nombre indica, implementa el método de integración numérica de *Runge-Kutta* con el cual se resuelve la ecuación de nuestro sistema. Se explicará con más detalle en el apartado 3.4 de esta memoria.

- **Clase Dibujo**

El cometido de esta clase es el apartado gráfico de nuestro proyecto. En ella están implementados los métodos que permiten realizar cualquiera de las representaciones gráficas (*Duffing* sin control, *Duffing* controlado, pandeo sin control y pandeo controlado). Además cuenta con otra serie de métodos auxiliares que se explicarán posteriormente.

A continuación se explican las cinco clases que se han utilizado y posteriormente se muestra el diagrama de clases (Figura 19) en el que se han suprimido los métodos de acceso a los atributos para aclarar su lectura.

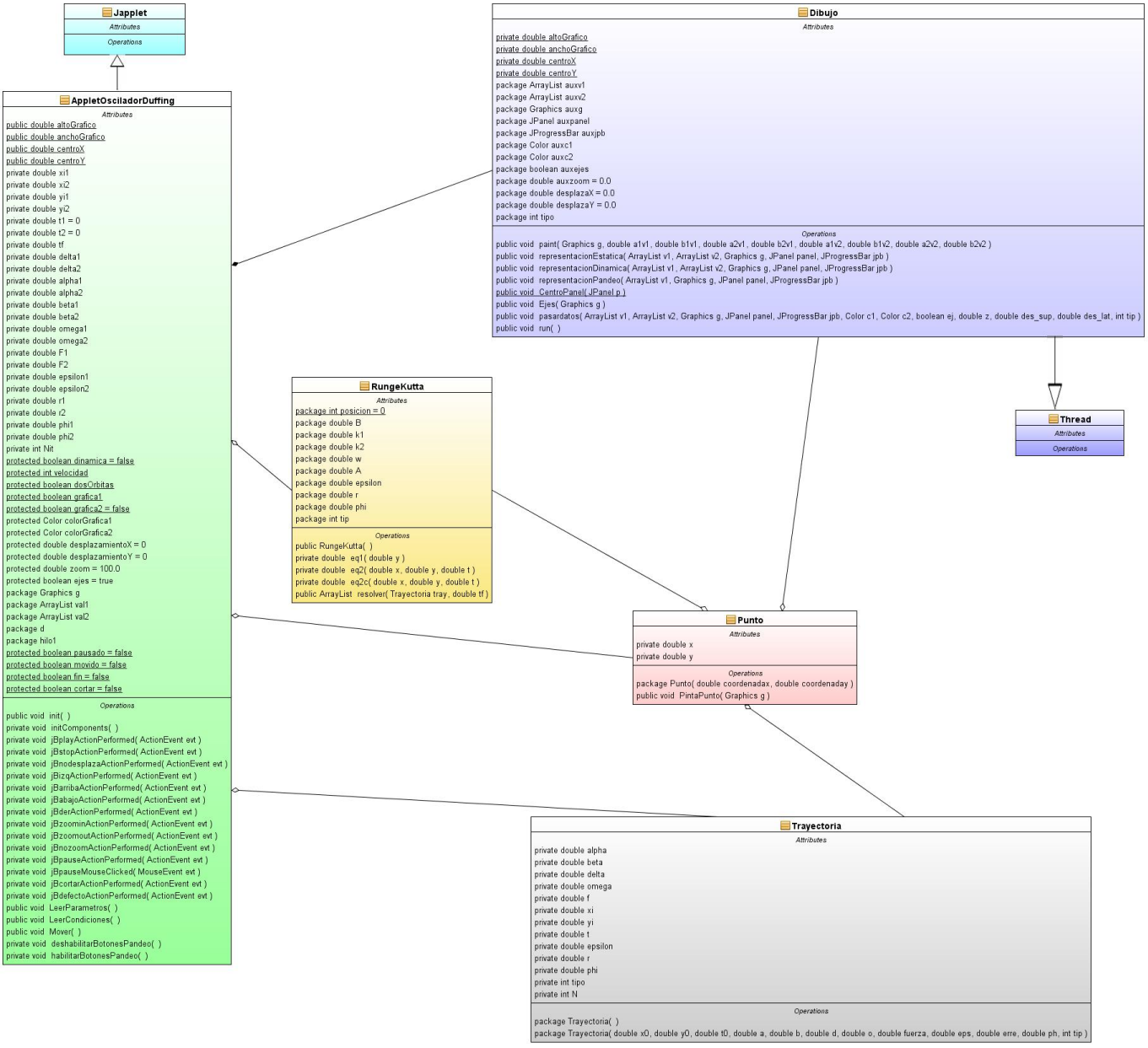


Figura 19.
Diagrama de Clases

3.3 Modo de uso

Cuando comenzamos a ejecutar nuestra aplicación, lo primero que debemos hacer es introducir las condiciones iniciales y valores que tomarán los diferentes parámetros en la ecuación. Estos valores serán utilizados sea cual sea el tipo de gráfico que queramos representar y que se elige mediante las pestañas de la parte superior derecha de nuestra interfaz. Si optamos por representar el *Duffing* (tanto controlado como no controlado), podemos elegir entre diferentes opciones. Además de los valores que hemos comentado anteriormente, podemos elegir el color con el que se dibujará la órbita. En este tipo de representación podemos dibujar dos órbitas cuyos parámetros y/o condiciones iniciales sean distintos. Otra de las opciones que se nos presentan es elegir si queremos dibujar los ejes *x,y* a la vez que la gráfica.

Por otro lado, tenemos que decidir si queremos una representación estática o dinámica (parte derecha de la aplicación), en la cual se vaya viendo la evolución de la/s órbita/s a lo largo del tiempo. En caso de decidirnos por una representación dinámica, podremos variar la velocidad con la que se dibuja mediante el indicador de velocidad. Además de ello, en la parte izquierda, en el panel "Opciones de la Gráfica", el botón de cortar nos permitirá borrar los puntos dibujados hasta el momento en el que se pulsa dicho botón. Tras su pulsación, la representación sigue su curso como antes.

Por otro lado aparecen los botones de Zoom y de Desplazamiento, que nos ayudarán tanto en la representación estática como dinámica para obtener una mejor visión de nuestra/s órbita/s y desplazarnos a lo largo de los ejes.

Si nos hemos decantado por la representación del pandeo de la viga, las opciones gráficas desaparecen. El estado de la viga (velocidad con la que oscila) cambiará en función de los valores que tome la ecuación. Esta representación sólo podrá ejecutarse de forma dinámica.

Independientemente del tipo de representación, será necesario el uso de los botones de acción. El botón *Play* iniciará la representación elegida, el *Pause* podrá utilizarse para interrumpir la representación dinámica en un momento que consideremos oportuno y el botón *Stop* limpiará el área de representación y restablecerá las opciones del gráfico. Si quisiésemos que las condiciones iniciales y los parámetros tomaran los valores por defecto, podemos utilizar el botón de la parte izquierda de nuestra interfaz.

3.4 Implementación

Es esta parte de la memoria se explican detalles correspondientes a la implementación del código de nuestro *applet.* Se irán explicando todas las clases que se han utilizado y sus métodos más relevantes.

- **La clase AppletOsciladorDuffing** es la clase principal de nuestro *applet* y se declara como una extensión de la clase de este tipo, y así. poder realizar las operaciones necesarias pertenecientes a esta clase.

Encabezado de la clase

```
public class AppletOsciladorDuffing extends javax.swing.JApplet {…}
```

Después del encabezado se produce la declaración de atributos y se implementan los métodos necesarios para el funcionamiento de nuestra aplicación.

El método init

```
public void init() {
        try {
            java.awt.EventQueue.invokeAndWait(new Runnable() {
                public void run() {
                    initComponents();
                }
            });
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
```

Este método se encarga de inicializar nuestro *applet.* En su interior podemos encontrar una llamada al método *initComponents()* que es donde se definen todos los objetos. A su vez, en el encabezado del método observamos que se utiliza la interfaz *Runnable,* que nos facilita el uso de la concurrencia, es decir, la posibilidad de ejecución de varios hilos simultáneamente.

El método jBplayActionPerformed

En este método se implementa la función del botón *Play* de nuestra aplicación. De forma resumida, podemos decir que leerá los parámetros y condiciones iniciales introducidas por el usuario para crear un objeto de *la clase Trayectoria*, para poder resolver la ecuación de nuestro sistema utilizando la

clase RungeKutta y dibujar mediante la *clase Dibujo* la representación elegida por el usuario.

```
private void jBplayActionPerformed(java.awt.event.ActionEvent evt) {

        jBplay.setEnabled(false);
        g=jLienzo.getGraphics();
        RungeKutta r = new RungeKutta();
        try {
                LeerCondiciones();
                LeerParametros();
                if (jCGrafica1.isSelected()){
                    grafica1=true;
                }
                if (jCGrafica2.isSelected()){
                    grafica2=true;
                }

                jBpause.setEnabled(true);

                Trayectoria tr1 = new
Trayectoria(xi1,yi1,t1,alpha1,beta1,delta1,omega1,F1,epsilon1,r1,phi1,
jCBTipoGrafica.getSelectedIndex());

                Trayectoria tr2 = new
Trayectoria(xi2,yi2,t2,alpha2,beta2,delta2,omega2,F2,epsilon2,r2,phi2,
jCBTipoGrafica.getSelectedIndex());

                val1 = r.resolver(tr1,tf);
                val2 = r.resolver(tr2,tf);

                d.pasarDatos(val1,val2,tr1,tr2, g,
jLienzo,jPBarraTiempo,colorGrafica1,colorGrafica2,ejes,zoom,
desplazamientoX,desplazamientoY,jCBTipoGrafica.getSelectedIndex());
                jPBarraTiempo.setMaximum(val1.size());
                hilo1 = new Thread(d);
                hilo1.start();

        } catch (Exception ex) {

Logger.getLogger(AppletOsciladorDuffing.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
```

El método jBstopActionPerformed

Con este método se detiene la ejecución de los procesos que estén activos, dejando libre el panel de representación para permitir que el usuario realice otra operación. Es importante pero su implementación no tiene ningún elemento destacable.

El método jBpauseActionPerformed

El objetivo de este método es permitir pausar cualquiera de las representaciones dinámicas de la aplicación. Si se está representando y se pulsa, el proceso se detendrá hasta que vuelva a ser pulsado el botón.

```
private void jBpauseMouseClicked(java.awt.event.MouseEvent evt) {
        pausado=!pausado;
        if (pausado){
            jBpause.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/jappletosciladorduffing/play
.png")));
        }
        else{
            jBpause.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/jappletosciladorduffing/paus
e.png")));
        }
}
```

Métodos de desplazamiento

```
private void jBizqActionPerformed(java.awt.event.ActionEvent evt) {
        desplazamientoX=desplazamientoX-10;
        Mover();
}
```

Métodos de zoom

```
private void jBzoominActionPerformed(java.awt.event.ActionEvent evt) {
        zoom=zoom*1.2;
        Mover();
}
```

Los siguientes métodos están relacionados entre sí. Como se mostró anteriormente, nuestro *applet* cuenta con una serie de botones de zoom y desplazamiento (sólo se muestra el código de desplazamiento hacia la izquierda y de ampliar zoom). Todos estos métodos, utilizan la operación auxiliar *Mover*, que se encarga de volver a dibujar el gráfica en la posición elegida por el usuario y con el tamaño seleccionado a través de la interfaz de la aplicación.

El método Mover

```
public void Mover(){
    movido=true;
    d.pasarDatos(val1,val2,tr1,tr2, g,
jLienzo,jPBarraTiempo,colorGrafica1,colorGrafica2,ejes,zoom,

desplazamientoX,desplazamientoY,jCBTipoGrafica.getSelectedIndex());    if
((!dinamica)||(fin)){
        hilo1 = new Thread(d);
        hilo1.start();
    }
    jLienzo.update(g);
}
```

Además de todos los métodos que han sido explicados anteriormente, dentro de esta clase nos podemos encontrar con otros que rigen las opciones de representación de las gráficas. Generalmente se encargan de controlar variables booleanas con las que se controlan las opciones y los tipos de representación.

Por otro lado existen también los métodos que se encargan del formato de cada uno de los elementos del *applet*.

- **La clase Trayectoria**

```
public class Trayectoria {…}
```

Esta clase se encarga de crear un objeto que aglutine las condiciones iniciales, los parámetros de las variables, el tiempo de integración y el tipo de gráfico elegido por el usuario, por lo que su único método interesante es el constructor de la clase.

```
Trayectoria(double x0,double y0,double t0,double a,double b,double d,double
o,double, fuerza,double eps, double erre,double ph, int tip){
        xi=x0;
        yi=y0;
        t=t0;
        alpha=a;
        beta=b;
        delta=d;
        omega=o;
        f=fuerza;
        epsilon=eps;
        r=erre;
        tipo=tip;
        phi=ph;
          }
```

- **La clase Punto**

```
public class Punto {…}
```

Se trata de una clase muy sencilla que cuenta solo con los atributos de tipo *double* (real en lenguaje matemático) y sus métodos de acceso. Por otro lado, cuenta con el método *PintaPunto*, que permite la representación gráfica de un punto a partir de sus dos coordenadas. El código implementado se muestra a continuación:

```
public void PintaPunto (Graphics g){
        g.drawLine((int)x,(int)y,(int)x,(int)y);
}
```

- **La clase RungeKutta**

```
public class RungeKutta {…}
```

Es la clase encargada de resolver las ecuaciones del sistema, atendiendo a un objeto de la clase *Trayectoria*, donde están almacenados los valores de condiciones iniciales, parámetros y tiempos de integración necesarios.

Para la resolución de la ecuación, se realiza un cambio de variable, con lo que surgen dos ecuaciones derivadas de la inicial. La primera de ellas queda implementada de la siguiente forma:

```
private double eq1(double y){
            return y;
}
```

En el caso de la segunda ecuación del sistema dependerá de si el usuario ha optado por la representación del *Duffing* sin control (eq2) o controlado (eq2c). Ambas ecuaciones se muestran a continuación:

```
private double eq2(double x,double y, double t){
      double sol;
     sol=A*java.lang.Math.sin(w*t)-(B*y)-(k1*x)-(k2*java.lang.Math.pow(x,
3));
     return sol;
}

private double eq2c(double x,double y, double t){
      double sol;
      sol=A*java.lang.Math.sin(w*t)-(B*y)-(k1*x)-
```

```
(k2*(1+epsilon*java.lang.Math.sin(r*w*t+phi))*java.lang.Math.pow(x, 3));
      return sol;
}
```

Las ecuaciones anteriores son utilizadas en el método principal de la clase. El *método resolver* sigue el algoritmo de cuarto orden y queda implementado en nuestro caso como sigue:

```
public ArrayList resolver(Trayectoria tray, double tf) throws IOException {

        ArrayList valores = new ArrayList();
        double xi=tray.getXi();
        double yi=tray.getYi();
        double t=0;
        k1=tray.getAlpha();
        k2=tray.getBeta();
        B=tray.getDelta();
        w=tray.getOmega();
        A=tray.getF();
        epsilon=tray.getEpsilon();
        r=tray.getR();
        tip=tray.getTipo();
        phi=tray.getPhi();

        double h=2.0*java.lang.Math.PI/500;
        double x=xi;
        double y=yi;

        Punto p0=new Punto(x,-y);
        valores.add(p0);
        double kx1=0,kx2=0,kx3=0,kx4=0,ky1=0,ky2=0,ky3=0,ky4=0;
        for (int i=1;i<(int)(tf/h);i++){

            t=t+h;
            if (tip%2==0){
                kx1=eq1(y);
                ky1=eq2(x,y,t);
                kx2=eq1(y+h/2.0*ky1);
                ky2=eq2(x+(h/2.0*kx1),(y+h/2.0*ky1),t+h/2.0);
                kx3=eq1(y+h/2.0*ky2);
                ky3=eq2(x+(h/2.0*kx2),y+(h/2.0*ky2),t+h/2.0);
                kx4=eq1(y+h*ky3);
                ky4=eq2(x+h*kx3,y+h*ky3,t+h);
            }
            else {
                kx1=eq1(y);
                ky1=eq2c(x,y,t);
                kx2=eq1(y+h/2.0*ky1);
                ky2=eq2c(x+(h/2.0*kx1),(y+h/2.0*ky1),t+h/2.0);
```

```
                kx3=eq1(y+h/2.0*ky2);
                ky3=eq2c(x+(h/2.0*kx2),y+(h/2.0*ky2),t+h/2.0);
                kx4=eq1(y+h*ky3);
                ky4=eq2c(x+h*kx3,y+h*ky3,t+h);
            }
            x=x+(h/6.0)*(kx1+2.0*kx2+2.0*kx3+kx4);
            y=y+(h/6.0)*(ky1+2.0*ky2+2.0*ky3+ky4);
            Punto p=new Punto(x,-y);
            valores.add(p);
            }
        return valores;
    }
```

Como puede observarse en el código, en función del control o no de nuestro sistema se ejecutaría la rama del *if* (*Duffing* sin control) o la del *else* (*Duffing* Controlado).

- **La clase Dibujo**

```
public class Dibujo implements Runnable {...}
```

Es la clase encargada de realizar todas las representaciones gráficas de nuestra aplicación. Como se puede ver en la cabecera, implementa la interfaz *Runnable* con la cual controlamos la concurrencia del sistema. Con ello podemos dibujar dos gráficas al mismo tiempo a una velocidad adecuada, sin que el sistema tenga que interrumpir ninguno de los procesos.

Antes de nada se recopilan los datos obtenidos de las demás clases utilizando el método *pasardatos* que se muestra a continuación.

```
public void pasarDatos(ArrayList v1,ArrayList v2,Trayectoria tray1,
Trayectoria tray2, Graphics g, JPanel panel, JProgressBar jpb, Color c1,
Color c2, boolean ej, double z, double des_sup, double des_lat, int tip) {
        auxv1 = v1;
        auxv2 = v2;
        auxg = g;
        auxpanel = panel;
        auxjpb = jpb;
        auxc1 = c1;
        auxc2 = c2;
        auxejes = ej;
        auxzoom = z;
        desplazaX = des_sup;
        desplazaY = des_lat;
        tipo=tip;
        tr1=tray1;
        tr2=tray2;
    }
```

Una de las operaciones más importantes de la clase es el método *run*, perteneciente a la clase *Runnable* de la cual ya hemos hablado y que se inicia cuando se ejecuta el método *start()* sobre algún objeto de esta clase *Dibujo*.

```
public void run() {
        try {
            if ((tipo==0)||(tipo==1)){
                if ((!AppletOsciladorDuffing.dinamica) ||
(AppletOsciladorDuffing.fin)){
                        this.representacionEstatica(auxv1,auxv2, auxg,
auxpanel, auxjpb);
                }
                else {
                        this.representacionDinamica(auxv1,auxv2, auxg,
auxpanel, auxjpb);
                }
            }
            else{
                representacionPandeo(auxv1, tr1, auxg, auxpanel, auxjpb);
            }


        } catch (IOException ex) {
            Logger.getLogger(Dibujo.class.getName()).log(Level.SEVERE,
null, ex);
        } catch (InterruptedException ex) {
            Logger.getLogger(Dibujo.class.getName()).log(Level.SEVERE,
null, ex);
        }
    }
      }
```

Cuando se lanza este método, lo primero que se hace es comprobar qué tipo de gráfico es el que ha elegido el usuario y así desencadenar las operaciones siguientes. A continuación se muestran dos métodos que podríamos llamar auxiliares:

El método CentroPanel

Mediante este método se calculan tanto las dimensiones del panel de dibujo como el centro de éste.

```
public static void CentroPanel(JPanel p) {
                  altoGrafico = p.getHeight();
                  anchoGrafico = p.getWidth();
                  centroX = anchoGrafico / 2;
                  centroY = altoGrafico / 2;
          }
```

El método Ejes

Este método se encarga de dibujar los ejes *x* e *y* en el panel de representación, así como los valores de éstos con una precisión de 0.5.

```
    public void Ejes(Graphics g) {

        g.setColor(Color.black);
        //Eje Y
        Font f = new Font("Arial",Font.PLAIN,9);
        Font f2 = new Font("Arial",Font.BOLD,12);
        g.setFont(f2);
        g.drawLine((int) (centroX + desplazaX), (int) (centroY +
desplazaY), (int) (centroX + desplazaX), 0);
        g.drawLine((int) (centroX + desplazaX), (int) (centroY +
desplazaY), (int) (centroX + desplazaX), (int) altoGrafico);
        g.drawString("Y", (int) ((centroX + desplazaX) + 4), 10);

        //Eje X
        g.drawLine((int) (centroX + desplazaX), (int) (centroY +
desplazaY), (int) anchoGrafico, (int) (centroY + desplazaY));
        g.drawLine((int) (centroX + desplazaX), (int) (centroY +
desplazaY), 0, (int) (centroY + desplazaY));
        g.drawString("X", (int) (anchoGrafico - 10), (int) ((centroY +
desplazaY) - 4));
        g.setFont(f);
        g.drawString("0.0", (int) (centroX + desplazaX), (int) (centroY +
desplazaY));

//Pintar Valores
        boolean fin = false;
        double punto = 0.5;
        while (!fin) {
            if ((punto * auxzoom > altoGrafico / 2) || (punto * auxzoom >
anchoGrafico / 2)) {
                fin = true;
            } else {
                g.drawString("-" + Double.toString(punto), (int) (centroX +
desplazaX), (int) ((centroY + desplazaY) - (int) (punto * auxzoom)));
                g.drawString("-" + Double.toString(punto), (int) (centroX +
desplazaX), (int) ((centroY + desplazaY) + (int) (punto * auxzoom)));
                g.drawString("-" + Double.toString(punto), (int) ((centroX
+ desplazaX) - (int) (punto * auxzoom)), (int) (centroY + desplazaY));
                g.drawString(Double.toString(punto), (int) ((centroX +
desplazaX) + (int) (punto * auxzoom)), (int) (centroY + desplazaY));
                punto = punto + 0.5;
            }
        }
}
```

Tras ésto, a continuación se muestra la implementación de los métodos que nos permiten dibujar las gráficas tanto de manera estática como dinámica. Ambos utilizarán el método *paint,* que será el que se encargue realmente de ir dibujando los puntos.

El método RepresentacionEstatica

Es el método encargado de la representación cuando el usuario ha decidido hacerlo de forma estática. Recorrerá los vectores de ambas órbitas en los cuales se han almacenado los valores resultantes de la ecuación, extrayéndolos y envainándoselos al método *paint,* que se encargará de dibujarlos.

```
public void representacionEstatica(ArrayList v1,ArrayList v2, Graphics g,
JPanel panel, JProgressBar jpb) throws IOException, InterruptedException {

        double xv1 = 0, yv1 = 0, xv2 = 0, yv2 = 0;
        double xantv1 = 0, yantv1 = 0, xantv2 = 0, yantv2 = 0;
        Punto pv1 = new Punto(0, 0);
        Punto pantv1 = pv1;
        Punto pv2 = new Punto(0, 0);
        Punto pantv2 = pv2;

        g = panel.getGraphics();
        CentroPanel(panel);
        if (auxejes) {
            Ejes(g);
        }
        pantv1 = (Punto) v1.get(0);
        xantv1 = pantv1.Getx();
        yantv1 = pantv1.Gety();
        pantv2 = (Punto) v2.get(0);
        xantv2 = pantv2.Getx();
        yantv2 = pantv2.Gety();
        int z=0;
        for (z = 1; z < v1.size(); z++) {

            pv1 = (Punto) v1.get(z);
            xv1 = pv1.Getx();
            yv1 = pv1.Gety();
            pv2 = (Punto) v2.get(z);
            xv2 = pv2.Getx();
            yv2 = pv2.Gety();
            this.paint(g, (xantv1) * auxzoom + centroX + desplazaX,
(yantv1) * auxzoom + centroY + desplazaY, xv1 * auxzoom + centroX +
desplazaX, (yv1 * auxzoom) + centroY + desplazaY,
```

```
                    (xantv2) * auxzoom + centroX + desplazaX, (yantv2) *
auxzoom + centroY + desplazaY, xv2 * auxzoom + centroX + desplazaX, (yv2 *
auxzoom) + centroY + desplazaY);
            xantv1 = xv1;
            yantv1 = yv1;
            xantv2 = xv2;
            yantv2 = yv2;

            }
        jpb.setValue(v1.size());
}
```

El método RepresentacionDinamica

Realiza las mismas operaciones que el método anterior, con una diferencia. Las órbitas se van dibujando poco a poco, atendiendo al valor de la velocidad que vaya eligiendo durante la ejecución el usuario (controlado mediante la operación *sleep* de los *threads*). Por otro lado, también existe la posibilidad de interrumpir la ejecución con el botón de *pause*. La barra de progreso irá aumentando a medida que el proceso de pintado de nuestra órbita va evolucionando. Además es necesario controlar aquellos momentos en los que el usuario decide utilizar los botones de zoom y desplazamiento durante el dibujado.

```
public void representacionDinamica(ArrayList v1,ArrayList v2, Graphics g,
JPanel panel, JProgressBar jpb) throws IOException, InterruptedException {
        double xv1 = 0, yv1 = 0, xv2 = 0, yv2 = 0;
        double xav1 = 0, yav1 = 0, xav2 = 0, yav2 = 0;
        double xantv1 = 0,yantv1 = 0,xantv2 = 0,yantv2 = 0;
        Punto pv1 = new Punto(0, 0);
        Punto pantv1 = pv1;
        Punto pantv2 = new Punto(0, 0);
        Punto pav2 = pantv2;

        g = panel.getGraphics();
        CentroPanel(panel);

        pantv1  = (Punto) v1.get(0);
        xantv1 = pantv1.Getx();
        yantv1 = pantv1.Gety();
        pantv2 = (Punto) v2.get(0);
        xantv2 = pantv2.Getx();
        yantv2 = pantv2.Gety();

        int i=0;
        int j=0;
        int progreso=i;
        for (i = 1; i < v1.size(); i++) {
```

```
            Thread.sleep((long) java.lang.Math.abs(((101-
AppletOsciladorDuffing.velocidad)*0.1)));
            progreso=i;
            Punto p1v1 = (Punto) v1.get(0);
            double xantav1 = p1v1.Getx();
            double yantav1 = p1v1.Gety();
            Punto p1v2 = (Punto) v2.get(0);
            double xantav2 = p1v2.Getx();
            double yantav2 = p1v2.Gety();
            jpb.setValue(i + 1);
            if (AppletOsciladorDuffing.cortar){

panel.getGraphics().clearRect(0,0,panel.getWidth(),panel.getHeight());
                if (auxejes) {
                    Ejes(g);
                }
                AppletOsciladorDuffing.cortar=false;
            }
            if (AppletOsciladorDuffing.movido){
                 for (j = 1; j < i; j++) {
//Representacion de los puntos ya representados
                        pantv1 = (Punto) v1.get(j);
                        xav1 = pantv1.Getx();
                        yav1 = pantv1.Gety();
                        pav2 = (Punto) v2.get(j);
                        xav2 = pav2.Getx();
                        yav2 = pav2.Gety();
                        this.paint(g, (xantav1) * auxzoom + centroX +
desplazaX, (yantav1) * auxzoom + centroY + desplazaY, xav1 * auxzoom +
centroX + desplazaX, (yav1 * auxzoom) + centroY + desplazaY,
                                (xantav2) * auxzoom + centroX + desplazaX,
(yantav2) * auxzoom + centroY + desplazaY, xav2 * auxzoom + centroX +
desplazaX, (yav2 * auxzoom) + centroY + desplazaY);
                        xantav1 = xav1;
                        yantav1 = yav1;
                        xantav2 = xav2;
                        yantav2 = yav2;
                        jpb.setValue(progreso);

System.out.println(AppletOsciladorDuffing.velocidad);
                }
                 AppletOsciladorDuffing.movido=false;
            }

                pv1 = (Punto) v1.get(i);
                xv1 = pv1.Getx();
                yv1 = pv1.Gety();
                pantv2 = (Punto) v2.get(i);
                xv2 = pantv2.Getx();
                yv2 = pantv2.Gety();
```

```
                this.paint(g, (xantv1) * auxzoom + centroX + desplazaX,
(yantv1) * auxzoom + centroY + desplazaY, xv1 * auxzoom + centroX +
desplazaX, (yv1 * auxzoom) + centroY + desplazaY,(xantv2) * auxzoom +
centroX + desplazaX, (yantv2) * auxzoom + centroY + desplazaY, xv2 *
auxzoom + centroX + desplazaX, (yv2 * auxzoom) + centroY + desplazaY);
                xantv1 = xv1;
                yantv1 = yv1;
                xantv2 = xv2;
                yantv2 = yv2;
                while (AppletOsciladorDuffing.pausado){
                //Esperamos sin hacer nada hasta que vuelva a pulsar
                        }

            }
        AppletOsciladorDuffing.fin=true;
}
```

El método Paint

Con este método se consigue pintar líneas que unen dos puntos consecutivos de cada una de las dos trayectorias que se pueden representar. Primero se mostrarían los ejes, si fueran necesarios, y a continuación los pares de puntos utilizando el color de gráfica seleccionado por el usuario.

```
public void paint(Graphics g, double a1v1, double b1v1, double a2v1, double
b2v1,
        double a1v2, double b1v2, double a2v2, double b2v2) throws
InterruptedException {
        if (((AppletOsciladorDuffing.dinamica)) && (auxejes)) {
            Ejes(g);
        }
        if (AppletOsciladorDuffing.grafica1){
            g.setColor(auxc1);
            g.drawLine((int) a1v1, (int) b1v1, (int) a2v1, (int) b2v1);
        }
        if (AppletOsciladorDuffing.grafica2){
            g.setColor(auxc2);
            g.drawLine((int) a1v2, (int) b1v2, (int) a2v2, (int) b2v2);
        }
}
```

El método RepresentacionPandeo

En este método se simula el pandeo de una viga, en el cual se toman las velocidades (*coordenada Y*) calculadas en la clase *Runge-Kutta*. Se trata de imágenes dinámicas en las cuales el usuario puede pausar dicha simulación. En cuanto al código, sólo hay que reseñar que el pandeo de la columna sería nulo

(la columna se queda parada en la posición de equilibrio) cuando la velocidad se acerca al 0 absoluto. Este método se utiliza de igual forma para mostrar el pandeo de la viga sin control o tras utilizar técnicas de control de la fase.

```
    public void representacionPandeo(ArrayList v1, Trayectoria tr,Graphics
g, JPanel panel, JProgressBar jpb) throws IOException,InterruptedException{

          int cargado=0;
          g = panel.getGraphics();
          CentroPanel(panel);
          Graphics2D g2d= (Graphics2D)g;
          Image fondo = new
ImageIcon(getClass().getResource("/jappletosciladorduffing/fondo.jpg")).get
Image();
          Image colc0 = new
ImageIcon(getClass().getResource("/jappletosciladorduffing/colc0.png")).get
Image();
          Image coli1 = new
ImageIcon(getClass().getResource("/jappletosciladorduffing/coli1.png")).get
Image();
          Image cold1 = new
ImageIcon(getClass().getResource("/jappletosciladorduffing/cold1.png")).get
Image();

          int controlarX=(int)centroX-280;
          int controlarY=(int)centroY-240;
          Punto p;
          double px,py,espera;
          boolean velocidadNula=false;
          boolean salir = false;

          p  = (Punto) v1.get(0);
          px = p.Getx();
          py = p.Gety();

          g2d.drawImage(fondo, (int)0, (int)0, panel);
          g2d.drawImage(colc0,controlarX,controlarY,panel);
          int k=1;
              while ((k<v1.size())&&(!salir)){
                 jpb.setValue(k);
                 p  = (Punto) v1.get(k);
                 px = p.Getx();
                 py = p.Gety();
                 espera=99*Math.abs(p.Gety());
                 if (pararPandeo(v1,tr)){
                     velocidadNula=((py>-0.001)&&(py<0.001));
                     k++;
                     if(velocidadNula){
                     salir=true;
                     }
                     while (AppletOsciladorDuffing.pausado){
                     //Esperamos sin hacer nada hasta que vuelva a pulsar
                     }
                     g2d.drawImage(fondo, (int)0, (int)0, panel);
                     g2d.drawImage(cold1,controlarX-60,controlarY,panel);
                     jpb.setValue(k);
                     p  = (Punto) v1.get(k);
```

```
                    px = p.Getx();
                    py = p.Gety();
                    espera=99*Math.abs(p.Gety());
                    velocidadNula=((py>-0.001)&&(py<0.001));
                     if(velocidadNula){
                        salir=true;
                    }
                    k++;
                    Thread.sleep((long) (espera+80));
                    while (AppletOsciladorDuffing.pausado){
                    //Esperamos sin hacer nada hasta que vuelva a pulsar
                        }
                    g2d.drawImage(fondo, (int)0, (int)0, panel);
                    g2d.drawImage(colc0,controlarX,controlarY,panel);
                    while (AppletOsciladorDuffing.pausado){
                    //Esperamos sin hacer nada hasta que vuelva a pulsar
                    }
                    g2d.drawImage(fondo, (int)0, (int)0, panel);
                    g2d.drawImage(coli1,controlarX-48,controlarY,panel);
                    jpb.setValue(k);
                    p  = (Punto) v1.get(k);
                    px = p.Getx();
                    py = p.Gety();
                    espera=99*Math.abs(p.Gety());
                    velocidadNula=((py>-0.001)&&(py<0.001));
                     if(velocidadNula){
                        salir=true;
                    }
                    k++;
                    Thread.sleep((long) (espera+80));
                    g2d.drawImage(fondo, (int)0, (int)0, panel);
                    g2d.drawImage(colc0,controlarX,controlarY,panel);
                    while (AppletOsciladorDuffing.pausado){
                    //Esperamos sin hacer nada hasta que vuelva a pulsar
                    }
                    k++;
                    jpb.setValue(v1.size());
                }
                else{
                    k++;
                    while (AppletOsciladorDuffing.pausado){
                    //Esperamos sin hacer nada hasta que vuelva a pulsar
                    }
                    g2d.drawImage(fondo, (int)0, (int)0, panel);
                    g2d.drawImage(cold1,controlarX-60,controlarY,panel);
                    jpb.setValue(k);
                    p  = (Punto) v1.get(k);
                    px = p.Getx();
                    py = p.Gety();
                    espera=99*Math.abs(p.Gety());
                    k++;
                    Thread.sleep((long) (espera+80));
                    while (AppletOsciladorDuffing.pausado){
                   //Esperamos sin hacer nada hasta que vuelva a pulsar
                        }
                    g2d.drawImage(fondo, (int)0, (int)0, panel);
                    g2d.drawImage(colc0,controlarX,controlarY,panel);
                    while (AppletOsciladorDuffing.pausado){
```

```
                    //Esperamos sin hacer nada hasta que vuelva a pulsar
                      }
                      g2d.drawImage(fondo, (int)0, (int)0, panel);
                      g2d.drawImage(coli1,controlarX-48,controlarY,panel);
                      jpb.setValue(k);
                      p  = (Punto) v1.get(k);
                      px = p.Getx();
                      py = p.Gety();
                      espera=99*Math.abs(p.Gety());
                      k++;
                      Thread.sleep((long) (espera+80));
                      g2d.drawImage(fondo, (int)0, (int)0, panel);
                      g2d.drawImage(colc0,controlarX,controlarY,panel);
                      while (AppletOsciladorDuffing.pausado){
                      //Esperamos sin hacer nada hasta que vuelva a pulsar
                      }
                      k++;
                      jpb.setValue(v1.size());
                  }
              }
    }
```

En la representación del pandeo de la viga nos hemos basado principalmente en el valor que ha tomado el forzamiento y en la variación de energías en los diferentes puntos de la trayectoria. Para ello hemos utilizado métodos auxiliares de los cuales hemos decidido que no era necesario detallar su implementación.

3.5 Movimientos representativos del sistema

Finalmente, mostraremos algunos tipos de movimientos representativos del sistema con los que mostraremos la utilidad de nuestro *applet*. Como ya se ha explicado anteriormente en esta memoria, nuestra aplicación nos permite realizar diferentes tipos de representación. Para comenzar, en el caso de representación del *Duffing* sin control, como *Duffing* controlado, se permite hacer una representación tanto de forma estática como dinámica, de una o dos órbitas simultáneamente. A continuación mostraremos una representación del *Duffing* sin control de forma estática de una órbita (Figura 20):

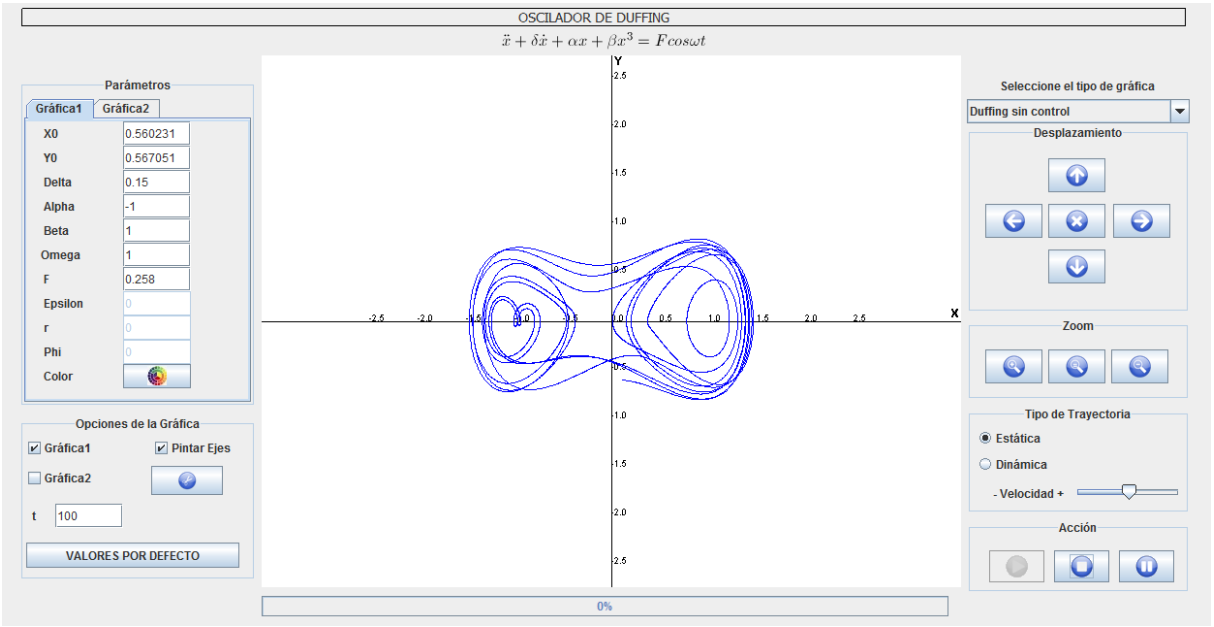


Figura 20.
Representación estática del Duffing sin control para una gráfica

A continuación, se muestra una representación dinámica de dos órbitas en las cuales se han introducido parámetros de control (Figura 21):

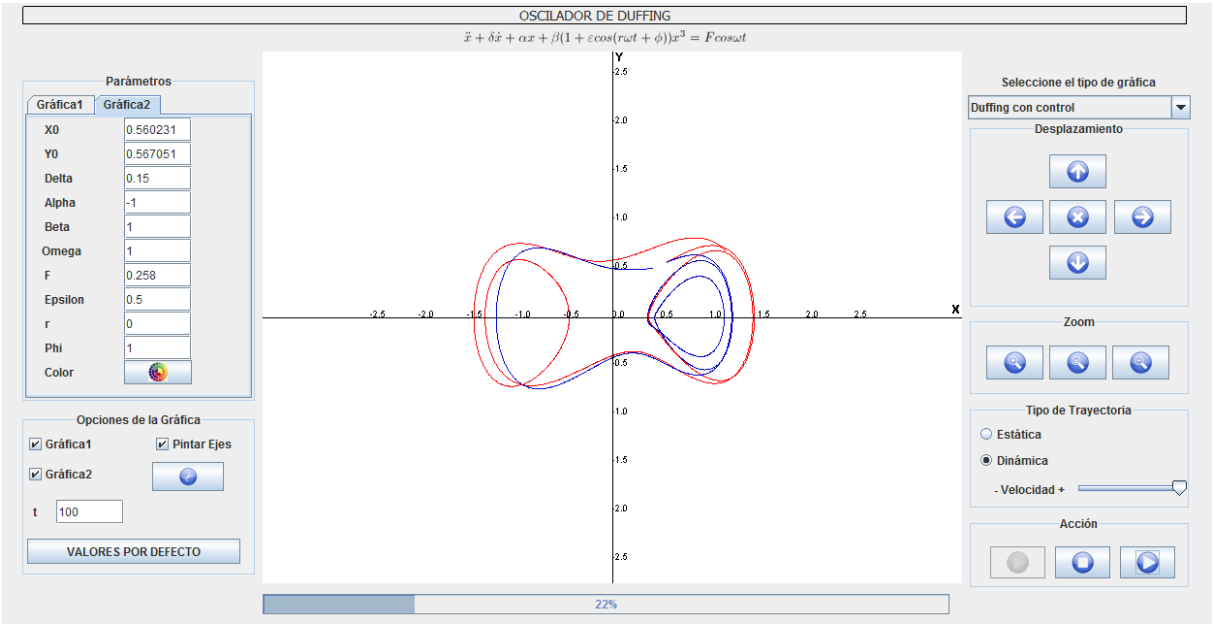


Figura 21.
Representación dinámica del Duffing controlado para dos gráficas al 22% de progreso

Por último, dos imágenes en las que se muestra el pandeo de la viga, en estado de equilibrio (Figura 22) y en una de las oscilaciones (Figura 23):

Figura 22.
Representación del pandeo de una columna en estado de equilibrio

Figura 23.
Representación del pandeo de una columna en una de las oscilaciones

4. Resultados y Conclusiones

A lo largo de todo este documento hemos pretendido explicar cómo se ha desarrollado todo el proyecto de la manera más sencilla posible. Se ha expuesto desde dos puntos de vista principalmente, el punto de vista físico, en el cual se ha detallado el sistema físico que hemos estudiado y la parte informática, en la que se han explicado aspectos tanto de diseño, como de implementación y uso de la aplicación creada.

Atendiendo a resultados conseguidos, tras el estudio previo del sistema del oscilador caótico de Duffing, hemos obtenido las siguientes conclusiones tras la realización de nuestro proyecto:

1. Hemos implementado del Oscilador Caótico de Duffing mediante un *Applet* de Java.

2. Hemos conseguido representar la trayectoria de dicho sistema tanto de una como de dos órbitas.

3. Nuestra aplicación permite la simulación del pandeo de una viga atendiendo a la ecuación de movimiento del sistema.

4. Hemos implementado la técnica del control de la fase para estabilizar nuestro sistema para regularizar su comportamiento.

4.1 Logros conseguidos

En cualquier proyecto informático, ya sea dentro del ámbito académico, o bien en el mundo laboral, se parte de una serie de requisitos que presenta el cliente, en nuestro caso, los tutores del proyecto. Esta serie de requisitos son aquellos detalles que debe cumplir la aplicación que se nos encarga. Todos estos requisitos fueron ya detallados en el apartado 3.1 Especificación, y ha sido posible su cumplimiento. A grosso modo, en este *applet* lo que se pretendía era la representación tanto de las gráficas como del pandeo de una viga según el sistema físico del oscilador caótico de Duffing, algo que se ha desarrollado satisfactoriamente.

Además de todo lo anterior, y fuera de los requisitos exigidos al inicio, se han incluido otras funcionalidades y herramientas al *applet* con las que se facilita su uso.

4.2 Posibles mejoras en la aplicación

Como en toda aplicación software, es importante que exista la posibilidad de incluir posibles mejoras en el futuro. En este caso, posiblemente se podrían incluirse otra serie de gráficos que ayudaran a la explicación y entendimiento de *sistema de Duffing,* como por ejemplo diagramas de bifurcación o un gráfico que represente la sección de *Poincaré*.

Por otro lado, existe la posibilidad de crear una aplicación genérica en la que se puedan agrupar diferentes sistemas físicos, o bien, integrarlos dentro de alguna página web dedicada a este tema.

Otra mejora, pese a que creo que es bastante fácil de utilizar, podría ser realizar algún cambio en la interfaz para mejorar la accesibilidad a los usuarios, además de incluir mensajes informativos.

5. Valoraciones Personales

He de reconocer que al principio del proyecto se me plantearon ciertas dudas sobre su realización. Había que conseguir crear una aplicación sobre un sistema físico en forma de *Applet de Java*, y personalmente, pensaba que no contaba con las nociones necesarias sobre ninguno de estos aspectos. En cuanto a la parte física, he aprendido bastante sobre el *Sistema de Duffing* gracias a las explicaciones de los tutores. Además del plano físico, hubo que recurrir a conocimientos matemáticos que nos ayudarán a la resolución de las ecuaciones de este sistema. Después llegaba la programación en Java. Era un reto personal. Durante la carrera no hemos contado prácticamente con asignaturas sobre este lenguaje, y he tenido que utilizar libros y apuntes para solucionar las muchas dudas que se creaban. Al final todo ha sido un reto que se ha conseguido superar satisfactoriamente y del que me siento muy orgulloso.

Otro de los aspectos que me ha parecido importante es que se ha asemejado en parte a todo aquello que vamos a encontrar en el mundo laboral. Un cliente (en nuestro caso, los tutores) nos pidieron la realización de una aplicación que trataba sobre un tema del cual no tenemos ningún conocimiento, cosa que nos pasará día tras día en cualquiera de las empresas que lleguemos a trabajar. Creo que esto será un punto de apoyo para esos momentos.

6. Bibliografía

[1] Edward Lorenz, *Deterministic nonperiodic flow*. Journal of Atmosphéric Sciences. Vol.20: 130-141, 1963.

[2] G. Litak, J.M. Seoane, S. Zambrano, M.A.F, *Non linear reponse of the mass-spring model with non-smooth stiffness*, International Journal of Bifurcation an chaos, (en prensa) 2011.

[3] G. Duffing, *Erzwungene Schwingungen bei Veränderlicher Eigenfrequenz*., F. Vieweg U. Sohn, Braunschweig, 1918.

[4] J. Guckenheimer and P. Holmes, *Nonlinear Oscillations, Dynamical Systems and Bifurcations of Vector Fields*, Springer-Verlag, 1983.

[5] E. Ott, C. Grebogi, J.A. Yorke, *Controlling Chaos, Physical Review Letters 64, 1196-1199,* 1990.

[6] S. Zambrano et al., *Numerical and experimental exploration of phase control of chaos, Chaos 16, 013111*, 2006.

[7] Richard L. Burden, J. Douglas Faires. Brooks – Cole Publishing, Numerical Analysis, 2004.

[8] José F. Vélez Serrano, Ángel Sánchez Calle, Alfredo Casado Bernárdez, Santiago Doblas Álvarez. *Técnicas avanzadas de diseño de software: Orientación a objetos, UML, patrones de diseño y Java*. Universidad Rey Juan Carlos.

[9] Tutorial sobre Applets, Java Sun. http://download.oracle.com/javase/tutorial/deployment/applet/index.html

[10] Tutorial sobre Threads y concurrencia, Java Sun. http://download.oracle.com/javase/tutorial/essential/concurrency/

[11] Tutoriales sobre Java Swing, Java Sun. http://download.oracle.com/javase/tutorial/uiswing/