ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Curso Académico 2011/2012

Proyecto de Fin de Carrera

Interfaz Visualizadora de Sistemas Físicos Caóticos

Autor: Juan Carlos Arqueros Vírseda

Tutores: Jesús Miguel Seoane Sepúlveda
Inés Pérez Mariño

Agradecimientos:

Me gustaría empezar estas menciones al apoyo mostrado, en primer lugar, a mis tutores y profesores del Departamento de Física de la URJC, Jesús Miguel Seoane e Inés Pérez. Por toda la dedicación y ayuda cada vez que les he visitado en su despacho y que han hecho que me interesara por la física y lo que le rodea.

En segundo lugar, a mi familia por el apoyo y la paciencia que han tenido mientras realizaba el proyecto. Mi madre sobre todo, por insistir semanalmente en darme ese pequeño "empujoncito".

En tercer lugar y no menos importante, a mis amigos por preguntar por mis adelantos con este trabajo y sobre todo a Ismael, por su ayuda en ciertas partes del mismo, orientándome y dándome aliento.

Y por último, a ese ex compañero que me abrió los ojos a Java y que supo en parte del proyecto que pasos debía dar.

Resumen:

Este proyecto, que presentaré en adelante, realiza la visualización de tres sistemas dinámicos. Estos tres sistemas (*Oscilador de Helmholtz*, *péndulo simple* y *péndulo doble*) con los que el usuario podrá interactuar serán visualizados gracias al lenguaje de programación *Java* y más concretamente a la tecnología de los *applet*.

Para la realización de esta aplicación *applet*, me he ayudado de dos programas. Uno para el modelado y diseño del interfaz (NetBeans) y otro para dar funcionalidad a esa interfaz y realizar cálculos, interacciones y visualizaciones (Eclipse).

Esta herramienta que he creado permitirá al usuario, mediante un navegador web poder visualizar las órbitas que varían en el tiempo de tres diferentes sistemas. Elegirá el que desee y mediante un interfaz sencillo podrá introducir valores a los parámetros y a las variables, reducir o aumentar la velocidad de la visualización de la onda, pausar o realizar un zoom, ver progreso en porcentaje, cuadricular el área de resultados, etc.

Índice

1. Introducción.. 7

2. Objetivos y metodología.. 11

2.1. Descripción del problema.. 12

2.2. Estudio de alternativas... 14

2.3. Metodología empleada... 17

3. Descripción informática... 19

3.1. Especificación.. 20

3.2. Diseño.. 21

3.3. Modo de uso.. 32

3.4. Implementación.. 33

3.5. Movimientos representativos del sistema.................................... 54

4. Resultados y conclusiones...................................... 58

4.1. Aplicaciones futuras... 59

5. Experiencia personal.. 60

6. Bibliografía.. 60

1. Introducción

Un *sistema dinámico* es un tipo de sistema que presenta un cambio o una evolución de su estado con el paso del tiempo.

En general, los sistemas dinámicos pueden clasificarse en lineales y no lineales dependiendo de la respuesta que presenten ante un estímulo externo. En los *sistemas lineales* el efecto producido es proporcional a la causa. Esto quiere decir que si perturbamos un sistema con una fuerza *F* y el efecto que producimos sobre ese sistema tiene valor *A*, si la fuerza fuera *2F* el efecto sería *2A*. O bien matemáticamente: $f(x) = Cx$, donde *C* es un constante de proporcionalidad.

Los *sistemas no lineales* son aquellos en los que el efecto no es proporcional a la causa. Estos sistemas son mucho más difíciles de analizar y, a diferencia de los lineales, pueden presentar un comportamiento errático o aperiódico, fenómeno conocido como caos, con comportamientos totalmente impredecibles a largo alcance.
El meteorólogo americano del MIT, Edward Lorenz (1917-2008) fue el primer científico que observó el comportamiento caótico en un sistema físico cuando estudiaba un modelo de predicción meteorológica [1]. En él observó como pequeñas variaciones en las condiciones iniciales producían que los estados finales respectivos fueran completamente diferentes. Esto es lo que se conoce como dependencia sensible a las condiciones iniciales que es el sello característico de un movimiento caótico. A esto se le llamó *Efecto Mariposa*, y se enunciaba diciendo que el aleteo de una mariposa en un punto podía tener consecuencias importantes en otro punto o lugar diferente. Se le llamó así después del experimento de Lorenz, ya que el sistema de Lorenz presenta una forma similar a la de las alas de una mariposa aleteando.

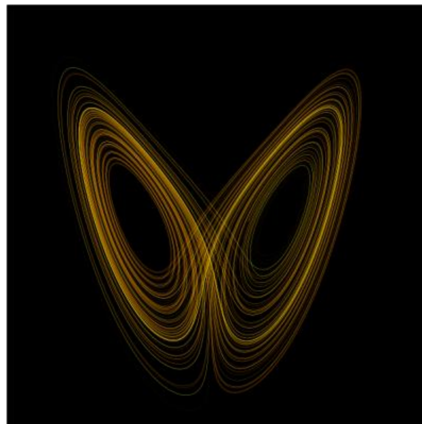


Figura 1. – Gráfico del sistema de Lorenz que recuerda a las alas de una mariposa

Para determinar de forma cuantitativa el comportamiento de los sistemas dinámicos es necesario integrar las ecuaciones del mismo, ya sea analíticamente o por métodos numéricos, por lo que se ha hecho imprescindible la ayuda de los ordenadores ya que gran parte de las ecuaciones diferenciales no se pueden resolver analíticamente.

Oscilador de Helmholtz:

Utilizaremos como modelo prototipo el oscilador de Helmholtz, que es el más simple oscilador no lineal con fugas o escapes. Para algunos valores de los parámetros de entrada, este oscilador presenta un valor crítico del forzamiento de todas las partículas, que escapan de su trayectoria normal. Usando la técnica de control de la fase, cambiando suavemente la forma del potencial a través de una perturbación periódica de la fase adecuada, se evitan las fugas en las diferentes regiones del espacio de fases. Este método puede ser útil para evitar fugas en las situaciones físicas más complicadas. Los sistemas dinámicos libres son típicos en la naturaleza.

En un sistema dinámico libre, hay una región del espacio de fases donde casi todas las trayectorias divergen de forma asintótica a infinito. Estas trayectorias han atraído una gran atención en el contexto de caos transitorio y, en particular, en los problemas de dispersión caótica, entre otros. El carácter generalizado de este tipo de sistemas dinámicos sugiere que hay situaciones en que podríamos estar interesados en evitar estas divergencias hasta el infinito, es decir, evitar escapes. Con el fin de definir lo que es un escape podemos imaginar el siguiente escenario. Supongamos que una partícula está bajo la influencia de algún objeto o potencial enorme. Bajo esta situación, decimos que un sistema dinámico tiene un escape siempre que esta partícula cruza un límite determinado y nunca regresa. También, desde un punto de vista práctico, será de gran interés poder controlar este tipo de comportamiento.

Desde el trabajo pionero del control del caos, debido a Ott, Grebogi, y Yorke (OGY) [7], se han propuesto esquemas de control que permiten obtener una respuesta deseada de un sistema dinámico mediante la aplicación de algunas pequeñas pero precisas perturbaciones seleccionadas. En este contexto, se han propuesto algunas técnicas que permiten evitar los escapes en sistemas dinámicos libres que presentan caos transitorio, con aplicaciones a situaciones diferentes en la física y la ingeniería.

Los métodos indicados para controlar el caos se pueden clasificar dependiendo de cómo interactúan con el sistema. Uno de los métodos de control del caos consiste en estabilizar las órbitas inestables utilizando pequeñas perturbaciones dependientes del estado del sistema. Sin embargo, en implementaciones experimentales, la respuesta rápida que estos métodos requieren no puede ser proporcionada por lo general. Para estas situaciones, existe un método más útil utilizado para suprimir el caos en sistemas dinámicos impulsados periódicamente.

Péndulo simple:

En este proyecto se estudia el sistema del *péndulo caótico*, siendo este un sistema dinámico no lineal que presenta comportamiento caótico para valores estándar de los parámetros. El estudio de los sistemas caóticos es un tema de gran

interés actual en Ciencia e Ingeniería. Un sistema que exhiba caos puede ser muy importante e incluso útil, pero también peligroso en muchos casos. Históricamente, las oscilaciones caóticas han sido un efecto a evitar en el diseño de cualquier tipo de dispositivo.

La idea intuitiva que se tiene de oscilación es la evolución en el tiempo de un sistema caracterizado por un movimiento de vaivén, ya sea éste regular o no. Existen multitud de situaciones en las que el movimiento oscilatorio es la dinámica predominante, siendo los ejemplos más comunes el de una masa suspendida en un muelle elástico, el péndulo de un reloj de pared, las oscilaciones de las cuerdas de una guitarra, etc. Sin embargo, existen otros ejemplos aparentemente menos familiares como son el comportamiento periódico de algunos ritmos biológicos tales como la respiración, las palpitaciones del corazón o fenómenos físicos que se repiten periódicamente como las mareas o las vibraciones de las moléculas. La evolución en el tiempo de estos sistemas puede ir desde un comportamiento totalmente periódico, como en los ejemplos anteriores, a un comportamiento irregular, en el que las oscilaciones nunca se repiten, denominado caótico.

El péndulo caótico surge cuando un péndulo es forzado periódicamente. En el caso de oscilaciones pequeñas el comportamiento es lineal y la aplicación de una fuerza externa periódica puede dar lugar a fenómenos de resonancia. Sin embargo, respuestas no lineales son inevitables en el mundo real. Por la ausencia de pautas de regularidad y por presentar un comportamiento donde una pequeña diferencia en las condiciones iniciales tiene efectos impredecibles, recibe el nombre de caótico.

A continuación, se describen algunos conceptos que son relevantes a la hora de hacer un análisis geométrico de un sistema físico. Uno de los conceptos fundamentales es el *espacio de fases*, que es una construcción matemática que permite representar el conjunto de posiciones y velocidades de un sistema de partículas. Si el movimiento de un sistema físico es periódico el sistema vuelve al mismo estado después de un ciclo completo. La representación de su trayectoria en el espacio de las fases es una curva cerrada. En cambio, si el movimiento de un sistema físico es caótico, en general, aparece una acumulación de puntos en el espacio de las fases que después de un largo período de tiempo, terminarían de llenarlo por completo. Estos dos tipos de comportamiento pueden verse en la figura 2.

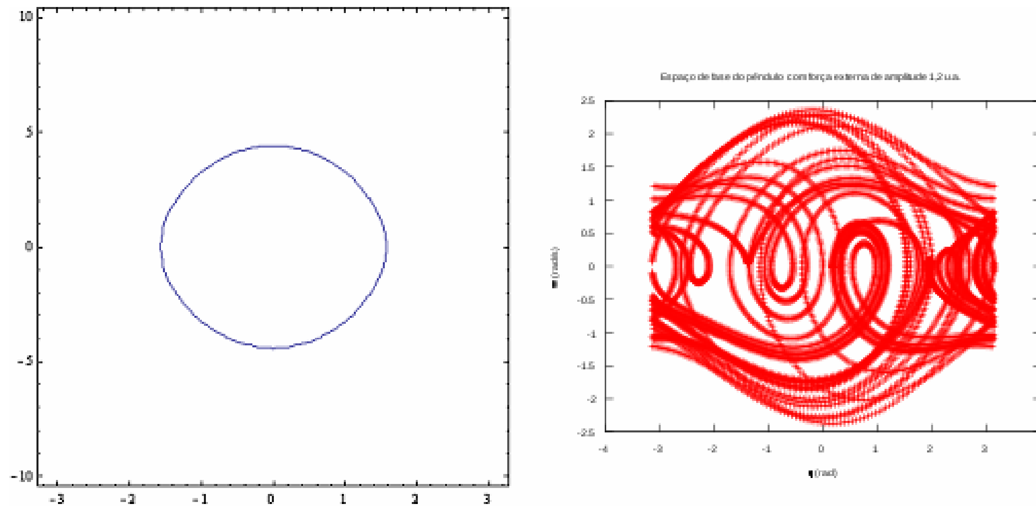


Figura 2.- Espacio de las fases del péndulo en régimen periódico (figura izquierda) y caótico (figura derecha)

Péndulo doble:

En este proyecto con el fin de contribuir al estudio de la teoría del caos y de la Dinámica No Lineal se ha simulado un sistema caótico paradigmático de un péndulo doble utilizando como herramientas las nuevas tecnologías. Aunque se haya oído hablar del movimiento caótico de un péndulo doble, quien no lo haya visto antes quedará cautivado por él. Y ésta no es una circunstancia aislada. Muchos sistemas mecánicos sencillos tiene una dinámica subyugante e incluso sorprendente; contemplarla puede captar la atención y avivar el interés por estudiar los sistemas con más detalle, aprendiendo entonces técnicas para analizar problemas más complicados y educando la intuición para elucidar la física del movimiento.

Sistemas mecánicos como el péndulo doble se pueden construir sin demasiadas complicaciones, pero sus características (como longitudes, masas, constantes elásticas…) no se pueden cambiar de manera inmediata y, obviamente, no es posible modificar la aceleración de la gravedad. Por este motivo, se ha desarrollado esta parte del proyecto que simule el movimiento del sistema en tiempo real.

En multitud de problemas aplicados, como es el caso del péndulo doble, donde las soluciones exactas no son posibles, se hace imprescindible usar métodos numéricos. La dinámica del péndulo doble viene dada por un conjunto de ecuaciones diferenciales que no se pueden resolver de modo exacto.

Este experimento es obviamente muy parecido al anterior, pero guarda más complejidad en su representación, ya que depende de dos mas y dos longitudes. Este péndulo muestra el comportamiento caótico resultante de tener un péndulo simple unido a la masa de otro péndulo simple.

En este caso no hay parámetros externos que condicionen el movimiento del péndulo para convertirlo en caótico. La influencia del primer péndulo sobre el segundo y viceversa nos dará buena muestra del comportamiento no lineal.

2. Objetivos y metodología

En nuestro proyecto lo más importante será la representación correcta de los tres sistemas dinámicos ofrecidos al usuario, la posibilidad de poder estudiar sus variaciones con respecto a cambios mínimos en los valores de los parámetros. Realmente puede llegar a ser el gran objetivo, pero habría que tener ciertos aspectos presentes para poder llegar a este resultado final.

- Parte de la labor de hacer un interfaz orientado al usuario es pensar directamente en él, pudiendo suponer que sea un neófito en física y sistemas complejos, un alumno o incluso un profesor. Por eso, esta interfaz con la que interactúe el usuario deberá ser fácil, cómoda y accesible. Los controles se basarán en la sencillez, tanto en la entrada de datos como en la representación final de cada sistema y todo se ajustará de la manera en que creí fuera más visual e intuitiva.

- En cuanto a la representación de los resultados, deberé decir que será lo más visual posible para mayor comodidad y claridad. Para ello se podrá decidir antes entre las dos posibles maneras de representación:

    1) **Representación estática**: Los resultados se presentan de manera directa sobre el panel dedicado a ellos sobre los ejes previamente mostrados.
    2) **Representación dinámica**: El trazado de puntos sobre los ejes se representa de manera continua evolucionando en base a la velocidad que se desee y controlando su ejecución con botones de pausa, reanudar o parar.

- Para todo esto haremos uso de la tecnología dada por Java mediante dos programas con los que poder construir código tanto para añadir interfaz y todos sus componentes(botones, cuadros de selección, campos de texto, paneles de visualización...) como para dar funcionalidad y darle a esa interfaz los mecanismos internos necesarios para que pueda llevarse a cargo esta tarea.

- Veremos más adelante como el código en Java está modularizado por clases que se encierran de paquetes que les da más singularidad con conjuntos de otras clases. Dentro de estas clases se describirán métodos que nos darán la

posibilidad de hacer uso para este "mecanismo interno" que manejara el *applet*.

- Además de todo esto, tendremos que hacer mención especial al uso de *threads* o hilos de ejecución que nos servirán para poder controlar las distintas ondas de un mismo sistema dinámico y la posibilidad de interrumpir o parar definitivamente cada una de ellas.

2.1. Descripción del problema

Como ya hemos dicho antes, nuestro proyecto engloba tres sistemas diferentes:

- **<u>Oscilador de Helmholtz:</u>**

  El oscilador de Helmholtz se describe con la ecuación siguiente:

$$\ddot{x} + \mu\dot{x} + \frac{dV}{dx} = F\cos(\omega t), \quad (1)$$

donde μ es el coeficiente de amortiguación, *V(x)* es la función potencial responsable de la fuerza de restauración que actúa sobre el sistema y *F cos(ωt)* es una fuerza periódica externa.

La idea principal de estos métodos sin "feedback" es aplicar una perturbación armónica tanto a algunos de los parámetros de sistema operativo del sistema como un forzamiento adicional, siendo su eficacia mostrada numérica y experimentalmente en diferentes trabajos.

- **<u>Péndulo simple:</u>**

  El péndulo caótico se describe mediante la siguiente ecuación:

$$m l^2 \frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + mgl\sin\theta = F\cos(\omega_t + \phi) \quad (2)$$

donde *m* es la masa del péndulo, l es la longitud del péndulo, *b* es el coeficiente de amortiguamiento o disipación (responsable de que las oscilaciones se atenúen con el tiempo), *F* es la amplitud de la fuerza externa aplicada, *ω* la frecuencia de dicha fuerza externa y ϕ el valor de la fase inicial. La variable θ representa la posición angular que ocupa el péndulo en cada instante. Este sistema presentará comportamiento periódico o caótico dependiendo del valor de los parámetros y de las condiciones iniciales.

- **Péndulo doble:**

El péndulo doble viene definido por estas dos ecuaciones:

$$\ddot{\vartheta}_1 = \frac{g(\sin\vartheta_2 \cos(\Delta\vartheta) - \mu \sin\vartheta_1) - (l_2 \dot{\vartheta}_2^2 + l_1 \dot{\vartheta}_1^2 \cos(\Delta\vartheta))\sin(\Delta\vartheta)}{l_1(\mu - \cos^2(\Delta\vartheta))},$$
$$\ddot{\vartheta}_2 = \frac{g\mu(\sin\vartheta_1 \cos(\Delta\vartheta) - \sin\vartheta_2) + (\mu l_1 \dot{\vartheta}_1^2 + l_2 \dot{\vartheta}_2^2 \cos(\Delta\vartheta))\sin(\Delta\vartheta)}{l_2(\mu - \cos^2(\Delta\vartheta))}. \quad (3)$$

donde *g* es la gravedad, ϑ_1 y ϑ_2 se refieren a las posiciones angulares de cada péndulo, y $\dot{\vartheta}_1$ y $\dot{\vartheta}_2$ son las velocidades angulares de cada uno de ellos . $\Delta\vartheta$ es el incremento de posiciones angulares de cada péndulo (ϑ_1-ϑ_2), μ es una variable que se puede sustituir por $1+(m_1/m_2)$ donde m_1 y m_2 representan las masas de cada péndulo; y por último l_1 y l_2 que serán las longitudes de cada péndulo.

Estas ecuaciones vienen como resultado de las ecuaciones de Euler-Lagrange que serían de esta forma:

$$\cos(\Delta\vartheta)\ddot{\vartheta}_1 + (l_2/l_1)\ddot{\vartheta}_2 = \dot{\vartheta}_1^2 \sin(\Delta\vartheta) - (g/l_1)\sin\vartheta_2,$$
$$\cos(\Delta\vartheta)\ddot{\vartheta}_2 + (\mu l_1/l_2)\ddot{\vartheta}_1 = -\dot{\vartheta}_2^2 \sin(\Delta\vartheta) - (g\mu/l_2)\sin\vartheta_1 \quad (4)$$

donde como antes vino explicado:

$$\Delta\vartheta \equiv \vartheta_1 - \vartheta_2 \qquad \mu \equiv 1 + (m_1/m_2)$$

2.2. Estudio de alternativas

Los movimientos que presentan los sistemas dinámicos lineales vienen descritos por ecuaciones diferenciales. Lo malo de estas ecuaciones dinámicas es que es difícil encontrar una solución parcial de manera analítica debido a la complejidad de éstas. Para poder conseguir una solución aunque aproximada se pueden utilizar varios métodos de integración numérica:

Método de Euler

El método de Euler (*Figura 3*) de primer orden es un método para la resolución de ecuaciones diferenciales ordinarias y las soluciones parciales que proporciona son aproximaciones numéricas. La idea de dicho método está basada en el significado geométrico de la pendiente de una función en un punto inicial dado para, posteriormente, poder hallar el punto siguiente.

Dicha ecuación es la siguiente:

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

donde *h* es el tamaño del paso temporal. Este método es más preciso cuanto menor sea el tamaño de paso o *h* y presenta un margen de error elevado respecto al verdadero valor de las funciones.

No es muy recomendable el uso de este tipo de método para realizar aproximaciones ya que no es muy preciso en comparación con otros métodos. Ese gran inconveniente es su poca precisión, ya que el error cometido en N pasos de integración es $N\,O(h^2) \approx O(h)$, y por tanto sólo decrece linealmente con la longitud del paso *h*.

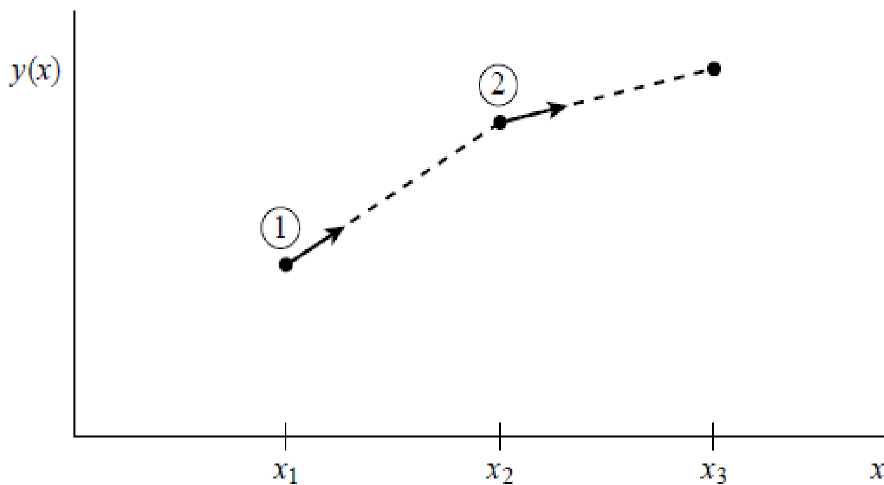


Figura 3.- Método de Euler, el inicio de cada intervalo se usa para encontrar el valor de la función en el siguiente intervalo.

Como solución se decidió mejorar aunque permitiera hacer aproximaciones para resoluciones de ecuaciones diferenciales. Esta resolución dejaba bastante que desear a medida que avanzábamos en los pasos de integración. Por eso surgió un método mejorado.

Método de Euler mejorado

La diferencia de este método (*Figura 4*) respecto al método anteriormente descrito consiste básicamente en usar la pendiente de la función en el punto inicial del intervalo para extrapolar el valor del punto siguiente (el paso n depende del n-1, el n+1 del n, y así sucesivamente). Por el contrario, en el método de Euler mejorado, dicho proceso no sera suficiente para hallar el valor del siguiente punto y se deberá hacer uso de un paso intermedio. Es decir, se debe hallar la derivada de la función en un punto intermedio del intervalo para, a continuación, extrapolar el valor en el punto final del mismo. Por dicho motivo, este método es también conocido como el método del punto medio.

Las ecuaciones del método serian las del siguiente conjunto:

$$k1 = hf(x_n, y_n)$$
$$k2 = hf(x_n + h/2, y_n + h/2\, k1)$$
$$y_{n+1} = y_n + k2 + O(h^3),$$

donde k1 y k2 serán dos cálculos intermedios sucesivos (k2 usa k1 e y_{n+1} usa k2) que darán como resultado una mejor aproximación que el método original de Euler.

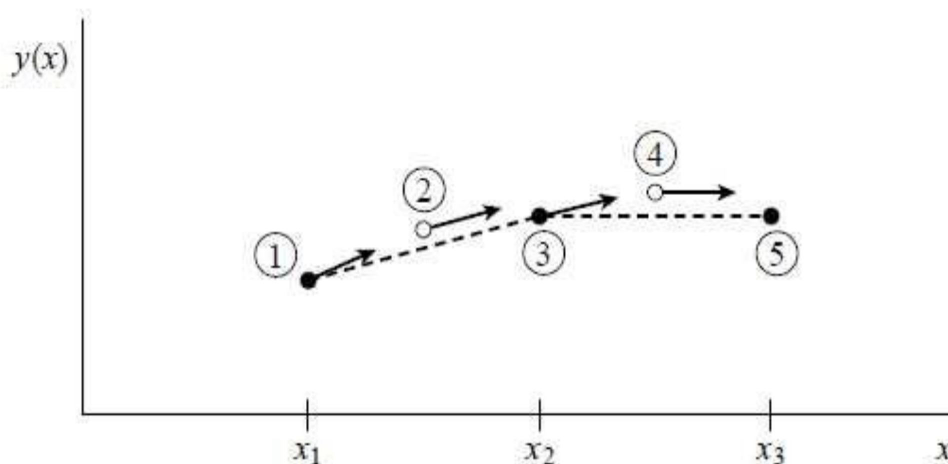


Figura 4.- Método del punto medio. Se aumenta la precisión usando el valor de la pendiente en el punto inicial (1) para encontrar los puntos intermedios de los intervalos (1,3) y (3,5). Una vez hallados los puntos intermedios 2 y 4, se usará la pendiente en ellos para poder hallar los valores finales de la función 3 y 5.

Método de Runge-Kutta

En sí, este método tiene dos singularidades con respecto a los dos anteriormente descritos: es un conjunto de métodos y se deriva del trabajo de Euler. No es un método en sí sino una importante colección de métodos iterativos implícitos y explícitos para aproximar las soluciones de ecuaciones diferenciales ordinarias. Usa

para cada paso cálculos intermedios que ayudan a disminuir el error final del siguiente paso (*Figura 5*). El método que veremos será el de cuarto orden, que en cada paso evalúa la derivada 4.

El conjunto de métodos Runge-Kutta [3] extienden la idea original de Euler al utilizar varias derivadas o tangentes intermedias, en lugar de solo una, para aproximar la función desconocida. Para realizar este método se deben calcular cuatro valores en cada intervalo k1, k2, k3 y k4, correspondientes a cuatro pendientes intermedias, y se debe elegir un paso de integración *h*.

Las ecuaciones del método de Runge-Kutta de cuarto orden que son las que elegiremos para nuestro problema son las siguientes:

$k1 = hf(x_n,y_n)$
$k2 = hf(x_n + h/2, y_n + k1/2)$
$k3 = hf(x_n + h/2, y_n + k2/2)$
$k4 = hf(x_n + h, y_n + k3)$
$y_{n+1} = y_n + k1/6 + k2/3 + k3/3 + k4/6 + O(h^5)$,

donde k1, k2, k3 y k4 serán cuatro cálculos intermedios sucesivos (k4 usa k3, k3 usa k2, k2 usa k1 e y_{n+1} usa todos los demás) que darán como resultado una mejor aproximación que los métodos de Euler.

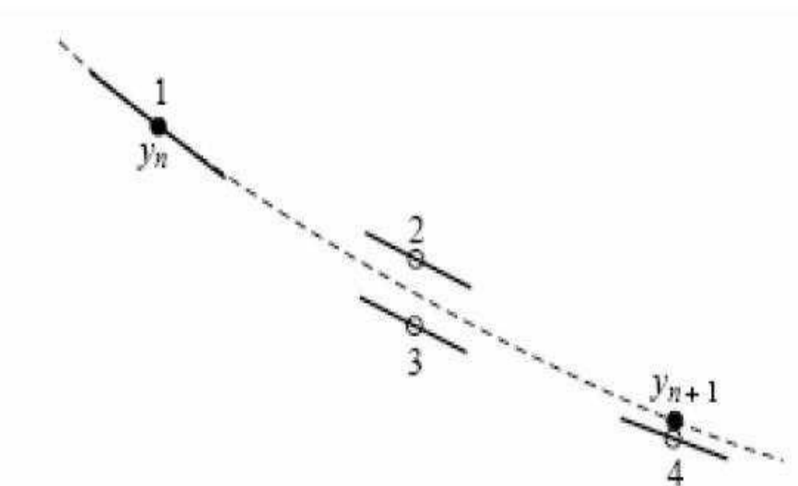


Figura 5.- Método de Runge-Kutta de cuarto orden. Para cada anchura de paso "h" se debe calcular la pendiente cuatro veces: una en el punto inicial (1), otra en los puntos intermedios (2,3) y otra en el punto final del intervalo (4). A partir de estas derivadas se calcula el valor final de la función (mostrado con un punto negro relleno)

2.3. Metodología empleada

Para la realización de cualquier trabajo englobado en la programación y en el desarrollo de una aplicación hay un modo de tomar el mando para no perder el control de en mitad del proceso y asegurar una serie de requisitos.

Nos centramos en dos posibles caminos para desarrollar la resolución del trabajo: metodología incremental y metodología en espiral. En nuestro caso y dado el aprendizaje autodidacta y lo que ello representa de cara a realizar un proyecto con un lenguaje de programación con el que se está poco familiarizado, hemos elegido el método en espiral (*Figura 6)*.

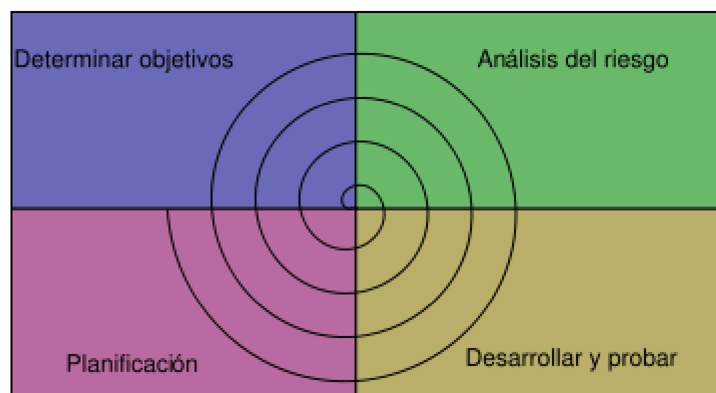


Figura 6.- Metodología en espiral

El modelo en espiral se basa en simplificar el desarrollo de sistemas complejos, de manera que iterativamente se van determinando los objetivos de la iteración, se analiza el riesgo que comportan, se desarrolla, se prueba, y finalmente se planifica la siguiente iteración, hasta realizar el software completo.

Para llevar a cabo el trabajo se deben cumplir una serie de pasos que cualquier programador, sea autónomo o perteneciente a un grupo de trabajo en una empresa importante, debe tener en mente:

Determinación de objetivos:

Se recogen las necesidades, requisitos y condiciones a satisfacer por el software en primer lugar o tras una o varias iteraciones de la "espiral". Esta primera fase marca el resto del desarrollo del interfaz ya que influirá en cómo haremos frente a las demás etapas. Para determinar esos objetivos es importante que quede todo claro, no haya conflictos dentro del proyecto, es decir, ninguna ambigüedad.

En este paso, se hará una visualización previa del diseño del applet con los componentes que deberá tener para cumplir con los requisitos y se irá modificando de cara a nuevas funcionalidades que se irán introduciendo una vez la fase "estándar" de ésta haya pasado por las tres etapas siguientes.

Planificación:

Se visualizarán las posibles soluciones que resuelvan el problema cumpliendo los requisitos de la primera fase. Así cuando se retome la parte de la programación podremos saber que métodos tendremos que realizar para que los engranajes interiores del applet se conecten e interactuen de la mejor manera. Para mostrar todo esto de un modo más claro adjuntaremos un diagrama de clases donde se verá como las clases comparten métodos.

Desarrollo y prueba:

Se codifica en lenguaje Java todas las funciones con sus diversos mecanismos internos dentro de cada clase independiente y cómo entre ellas existe una serie de relaciones para poder llevar a cabo la fase de planificación anterior. Se desarrolla código interno de los métodos, uso de atributos, desarrollo de “listeners”...

Cada acción tiene una reacción y hay que tener en cuenta que cuanto más crece el código, el número de interacciones entre clases también aumenta. Esto puede llegar a ser un problema, porque un mísero cambio en uno de esos métodos trae consigo un cambio en las llamadas a éstos desde otras clases, con lo que hay tener especial cuidado.

Análisis de riesgos:

Analizamos el código hecho y si cumple todo lo que de momento hemos ido desarrollando con los requisitos iniciales, iremos por buen camino. En caso de que no fuera así, tendríamos que hacer una subfase dentro de ésta que consistiría en depurar el código para llegar a esa fase parcial que cumpla con los objetivos. Después de esta fase habremos completado un ciclo y si la solución es parcial deberemos realizar otro para estar más cerca de la solución final buscada.

3. Descripción informática

Para poder empezar a describir cómo resolveros todos los problemas informáticos que nos darán un resultado óptimo para la visualización debemos empezar por contar en qué consiste un applet.

Esta tecnología basada en el lenguaje de programación Java, permite cargar en el navegador de cualquier usuario la aplicación que tenemos subida en el servidor del

que ofrece ésta. Nuestra aplicación es cargada en una página web mediante un sencillo código HTML que la invoca.

Esto es posible gracias a la tecnología de la máquina virtual de Java, que permite que desde cualquier navegador (Firefox, Internet Explorer, Chrome...) que soporte el plug-in de Java, nos de acceso a la aplicación.

Algunas características que distinguen a los applets son:

- Multiplataforma y portable. Los applets permiten ser cargados en cualquier plataforma (Windows, Linux, Mac OS) que disponga de la tecnología de Java (JVM). Además, la anterior característica antes mencionada de poder usarla en los navegadores que soporten la máquina virtual de Java.

- Seguridad. Los applets tienen la facultad de restringir el acceso a partes sensibles que no quieran ser modificadas o borradas. Sólo si el programador permite esta edición, el usuario podrá tener acceso a esos privilegios.

3.1. Especificación

En esta parte del desarrollo de nuestra aplicación debemos dar los objetivos que queremos que nuestra aplicación cumpla al finalizar el código. Debemos discernir los requisitos del proyecto para poder llegar a una mejor conclusión del proyecto.

Requisitos funcionales:

- La aplicación deberá mostrar los resultados en función de los datos introducidos por el usuario.

- La aplicación deberá mostrar los resultados para cualquiera de los sistemas planteados en él.

- La aplicación deberá dar la opción al usuario para poder visualizar el contenido en dos idiomas: español e inglés.

- La aplicación deberá dar la opción al usuario de mostrar los resultados de manera dinámica y estática.

- La aplicación deberá dar opción al usuario de poder dar 3 diferentes series de resultados visuales con parámetros distintos.

- La aplicación deberá dar opción al usuario para poder cambiar el color de cada una de las representaciones distintas dentro de cada sistema.

- La aplicación deberá dar la opción de guardar e imprimir los resultados mostrados por pantalla.

- La aplicación deberá dar la opción al usuario para el control de la animación de manera dinámica para poder pausarla, reanudarla o resetear los valores.

- La aplicación deberá dar opción al usuario para poder cortar el curso del dibujo dinámico de una gráfica.

- La aplicación deberá dar la opción al usuario para poder moverse sobre la imagen y hacer zooms para alejar o acercar los resultados.

- La aplicación deberá dar opción al usuario para poder cuadricular la imagen de los resultados o para poder numerar los ejes.

- La aplicación deberá dar opción al usuario para variar la velocidad de representación de las animaciones.

- La aplicación deberá dar opción al usuario para poder cambiar de sistema en cualquier momento.

Requisitos no funcionales:

- La aplicación deberá ser desarrollada en un applet de JAVA.

- La interfaz de la aplicación deberá ser atractiva visualmente para el usuario.

- La interfaz de la aplicación deberá ser fácil de usar e intuitiva.

- La aplicación deberá responder a los órdenes de usuario efectiva y rápidamente.

- La aplicación debe ser robusta y no permitir la introducción de valores erróneos.

3.2. Diseño

Para poder llevar a cabo el proyecto hemos utilizado el entorno de desarrollo integrado (IDE) de NetBeans. NetBeans es una herramienta pensada para escribir, compilar, depurar y ejecutar programas. Además, en nuestro caso nos servirá para la definición gráfica de nuestro applet. Con su entorno de construcción gráfico será muy fácil crear las “cajas” o JPanel que contengan el resto de elementos que conforme nuestro applet. Gracias a NetBeans tendremos la base para poder programar cada uno de los componentes que con él hemos creado.

Para la escritura de código, darle aplicación a botones, cajas de texto, menús desplegables y demás hemos usado la herramienta Eclipse. Da mejor flexibilidad para poder corregir código y permite una depuración del mismo, muy fácil e intuitiva.

Para poder realizar todo lo que nos requería el applet, Java nos da dos paquetes gráficos: Awt y Swing. Tanto una como otra serían de gran ayuda, pero Swing nos dará más serie de componentes y controles gráficos para el proyecto. Por eso mismo, haremos uso únicamente del paquete Swing. Además este paquete nos ofrece una serie de clases muy diversas. Ahora mostraremos cada una de las que han sido usadas para el applet:

- *JButton:* Es un componente que reacciona al pulsar encima de él para realizar acciones. En nuestro caso tenemos gran cantidad de botones, desde los colores para las ondas, pasando por los que nos mueven por la gráfica, hasta los que realizan zoom-in y zoom-out en la gráfica.

- *JPanel:* Es un componente que sirve de contenedor para otros componentes, incluso más JPanels. Sirve para agrupar los elementos importantes y para dar cohesión a todos ellos. En nuestro caso, servirá para agrupar cada ecuación y para mostrar los resultados en gráficas, entre otras cosas.

- *JToogleButton:* Es una clase similar a la JButton (de hecho forman parte de la clase abstracta AbstractButton) que permite tener la funcionalidad de botones presionados, un ejemplo de esto, son los botones de Negrita, Cursiva, Subrayado. Los usaremos para los botones de cuadrícula, paso de estático a dinámico e idioma.

- *JComboBox:* Es un componente que nos permite elegir entre una serie de valores definidos de antemano. Lo hemos usado para seleccionar las ecuaciones que queremos visualizar y para elegir el grosor para representarlas.

- *JLabel:* Es un componente muy básico. Son etiquetas de texto, aunque en ellas se pueden cargar imágenes. En nuestro caso caso las usamos para saber que caja de texto corresponde a cada variable y para cargar el icono del proyecto, por ejemplo.

- *JTabbedPane:* Es un componente que nos da la funcionalidad para cambiar entre varios componentes, como por ejemplo JPanel. En el proyecto usaremos este componente para cambiar entre las diferentes ondas dentro de cada ecuación para poder hacer varias representaciones de la misma.

- *JTextField:* Es un componente que permite introducir texto al usuario. En nuestro caso, lo usamos para introducir datos para los valores iniciales y para los parámetros.

- *JProgressBar:* Es un componente que muestra al usuario una barra de progreso. Para nuestro caso es obvio que lo usaremos para ver en que punto de la representación está la gráfica.

- *JScrollBar:* Es un componente deslizable que según en que punto este el marcador, da un valor u otro. Para nuestro *applet* los hemos utilizado para variar la velocidad de representación del modo dinámico.

3.2.1. Menú de elección de ecuación:

Es un elemento llamado JComboBox (*Figura 7*) que nos permite elegir qué ecuación queremos que se visualice en el panel de ecuación.

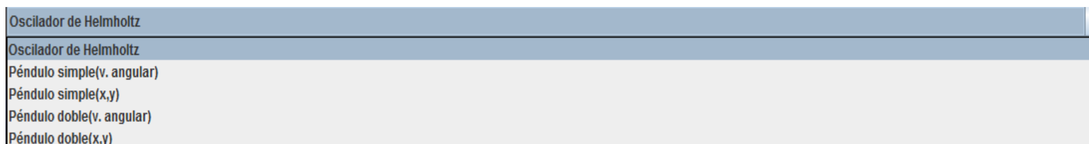


Figura 7.- Menú de selección de ecuación

3.2.2. Panel de ecuación:

Es la ventana principal donde se nos va a mostrar el interfaz de usuario exclusivo para cada ecuación (*Figura 8*). En él iremos viendo cada uno de los distintos componentes y contenedores que los mantienen.

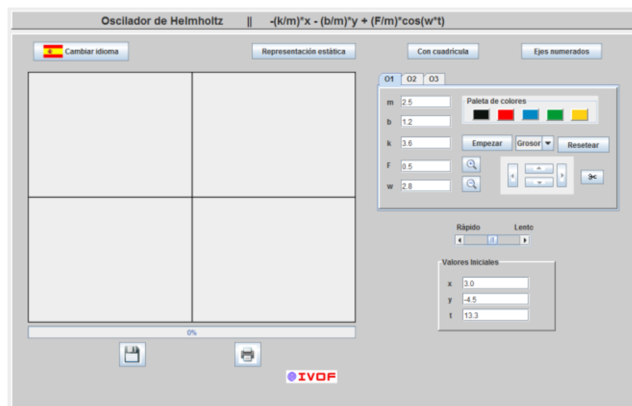


Figura 8.- Panel de ecuación, en concreto el Oscilador de Helmholtz

3.2.2.1. Panel de ondas:

En realidad no es un panel únicamente lo que se muestra, sino un compendio de tres paneles unidos bajo la clase JTabbedPane (*Figura 9*) que los agrupa y diferencia por pestañas.

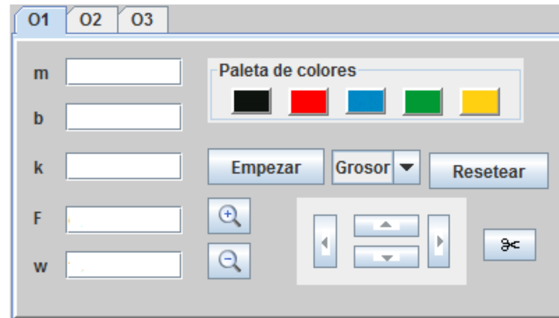


Figura 9.- Panel de las tres órbitas

En esta imagen podemos ver claramente las partes que contienen el panel. Por un lado y a la izquierda siempre, sea cual sea la ecuación, los parámetros. A la derecha tendremos herramientas para una vez calculados los resultados poder controlarlos y editarlos: paleta de colores, herramientas de desplazamiento sobre la gráfica, botón de empezar/pausar/reanudar y resetear, corte de gráfica y elección de grosor.

3.2.2.1.1. Paleta de colores:

Es un panel que nos muestra cinco botones (*Figura 10*) para poder elegir el color que queramos para las gráficas. La única imposición que se hace es que hay que elegirlo antes de pulsar el botón de empezar. Nos da a elegir entre negro, rojo, azul, verde y amarillo.

Figura 10.- Paleta de cinco colores

3.2.2.1.2. Panel de desplazamiento:

Contiene una serie de botones (*Figura 11*) para poder desplazarnos sobre las gráficas en el momento que deseemos, es decir, si han terminado su ejecución o incluso mientras se lleva a cabo su pintado (modo dinámico).

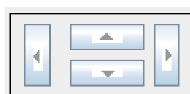


Figura 11.- Panel de desplazamiento

3.2.2.2. Control de velocidad:

Es un componente llamado JScrollBar (*Figura 12*) que permite hacer un deslizamiento de un punto sobre una barra para definir cierto valor único. En nuestro caso es perfecto para dar la velocidad a la representación dinámica del dibujo.

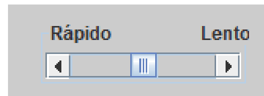


Figura 12.- Control de desplazamiento para la velocidad

3.2.2.3. Panel de valores iniciales:

Este panel nos enseña las cajas de texto para los valores iniciales que tengan cada una de las ecuaciones. Enseñaremos dos tipos de paneles debido a la diversidad de ecuaciones que podemos mostrar en el proyecto.

La más sencilla (*Figura 13*) nos muéstralos valores iniciales de *x, y* y el tiempo:

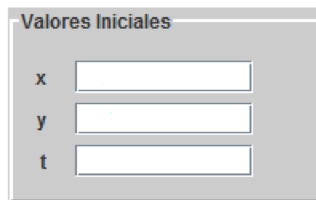


Figura 13.- Panel de valores iniciales. En este caso para Oscilador de Helmholtz y péndulo simple

La otra opción (*Figura 14*) del panel de valores iniciales tendría más campos de texto. El péndulo doble necesita dos valores iniciales para la posición angular y dos más para la velocidad angular correspondientes a cada péndulo. Además, como en el caso anterior, necesitamos un valor para la *t*.

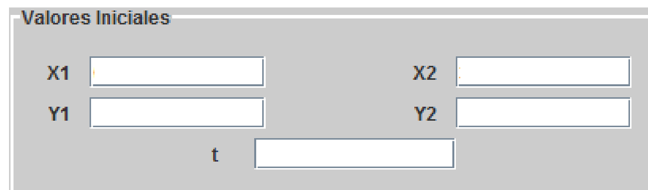


Figura 14.- Panel de valores iniciales. En este caso para el péndulo doble

3.2.2.4. Botón de cambiar idioma:

Nos da la opción de poder cambiar el idioma (*Figura 15*) de todo el *applet* entre el idioma estándar (español) e inglés. Hemos introducido además por comodidad un icono con la bandera española e inglesa.

Figura 15.-Botón para cambiar de idioma

3.2.2.5. Botón de representación:

Con este botón podemos cambiar el modo de representación de gráficas (*Figura 16*). Por un lado el estático o directo, y por otro el dinámico o secuencial.

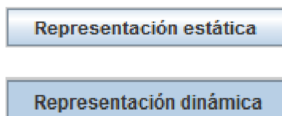


Figura 16.- Botón para cambiar el modo de representación

3.2.2.6. Botón de cuadrícula:

Este botón permite el colocar la “rejilla” de cuadrícula (*Figura 17*) para poder observar mejor que valores van tomando cada gráfica. Puede activarse o desactivarse cuando se desee, salvo si está pulsado el botón para numerar los ejes que a continuación explicaremos.

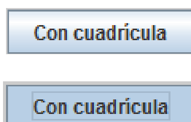


Figura 17.- Botón para activar la cuadrícula

3.2.2.7. Botón de numeración de ejes:

Este botón permite al usuario numerar los ejes (*Figura 18*) para poder interpretar mejor los valores que vayan representando las gráficas. Puede activarse o desactivarse cuando se quiera, salvo si está pulsado el botón anteriormente explicado: botón de cuadrícula.

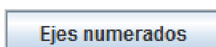


Figura 18.- Botón para activar la numeración de ejes

3.2.2.8. Botón de guardado:

Este botón nos permite guardar la imagen generada (*Figura 19*) sobre el panel de resultados en formato JPG en la carpeta de nuestro disco que deseemos.

Figura 19.- Botón de guardado del panel de representación y cuadro para el guardado

3.2.2.9. Botón de impresión:

Este botón nos permite imprimir la imagen (*Figura 20*) generada sobre el panel de resultados.

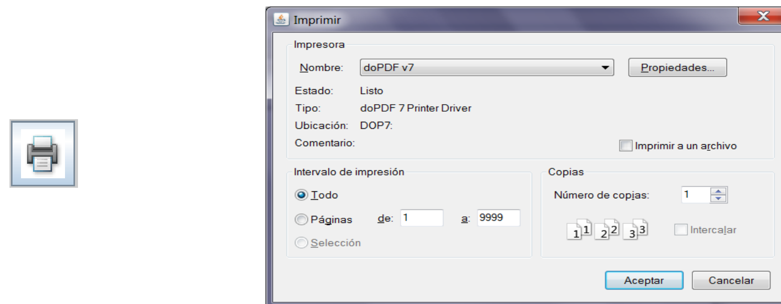


Figura 20.- Botón de impresión del panel de representación y cuadro de impresión

3.2.2.10. Barra de progreso:

Esta barra nos muestra el progreso de las ejecuciones (*Figura 21*). En la representación estática sólo tendrá dos posibles estados: 0% y 100%. Sin embargo, en la ejecución dinámica, se podrá dar el caso de porcentajes intermedios y la actualización de la barra de progreso según se vayan introduciendo mas ondas de la misma ecuación en la ejecución.

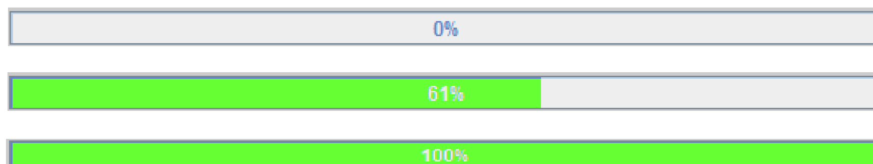


Figura 21.- Barras de progreso

3.2.2.11. Panel de resultados:

Este panel nos muestra los resultados (*Figura 22*) obtenidos a través de los cálculos de *Runge-Kutta* de los valores iniciales y los parámetros introducidos por el usuario en el sistema.

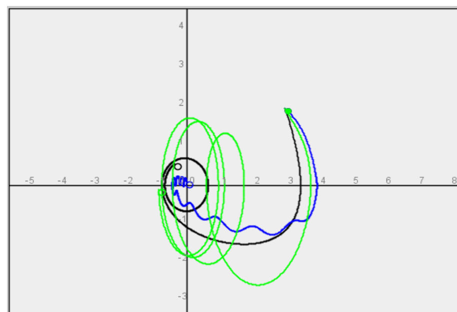


Figura 22.- Panel de representación de las ecuaciones

3.2.2.12. Icono de proyecto:

Es una etiqueta a la que se le ha cargado una imagen GIF (*Figura 23*). No muy importante en otro caso, pero muestra el icono dado para este proyecto llamado IVOF

Figura 23.- Icono del applet IVOF

3.2.3. Enlace web:

Mediante un JLabel o etiqueta y una librería especial que permite el enlace hacia páginas web podemos mostrar este link (*Figura 24*). Funciona como un botón que al ser pulsado nos abre una página en nuestro navegador por defecto hacia la web del Departamento de Física de la Universidad Rey Juan Carlos I.

Figura 24.- Enlace a la página del Departamento de Física de la URJC

3.2.4. Diseño de clases:

En este apartado haremos un diagrama de clases (*Figura 25*) de todas las que componen el applet. Por motivos de espacio y claridad hemos decidido con ciertos atributos (ptijeras1, ptijeras2, ptijeras3; por ejemplo) y con algunos métodos (botonUp1ActionPerformed, botonUp2ActionPerformed, botonUp3ActionPerformed; por ejemplo) unirlos bajo un sólo atributo o un sólo método (ptijerasN y botonUpNActionPerformed).

Además como tenemos cinco tipos de ecuaciones a mostrar, necesitaremos cinco tipos distintos de paneles contenedores para ellas. Por ello, en este problema también hemos decidido resumir ese problema para una clase tipo de una ecuación.

Como es de entender hemos decidido mostrar las clases más importantes referentes al proyecto, dejando algunos que son de apoyo, para explicarlas más adelante.

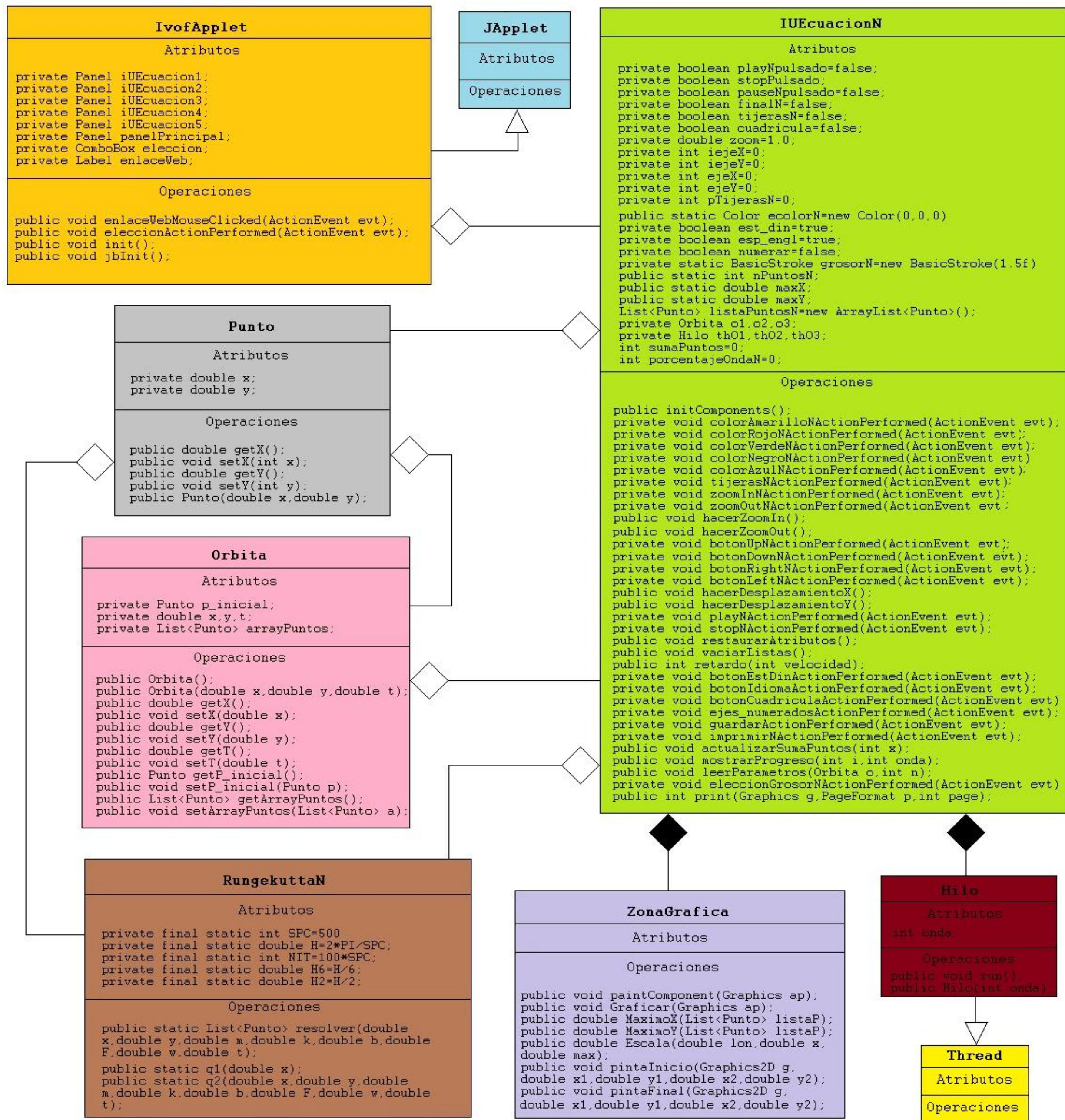


Figura 25.- Diagrama de clases

Clase IvofApplet:

Esta clase compone el primer paso del applet. Esta clase hereda de la interfaz Japplet y es la clase que sirve para que las demás clases tengan cabida y den su funcionamiento óptimo al proyecto. Sobre ella crearemos los cinco paneles donde se cargará cada clase IUEcuacionN que están conectadas a ésta mediante una relación de agregación de composición. Además, en esta clase también también tendremos otros dos componentes: JComboBox que nos dará la opción de elegir la ecuación a visualizar, y un JLabel que nos servirá de enlace a la página web del Departamento de Física de la URJC.

El método *init()* contenido en esta clase se encarga de arrancar el applet y cargar el método *jbInit()* que cargará los elementos anteriormente descritos sobre el applet.

Clase IUEcuacionN:

Aunque la clase *IvofApplet* sea la que sirva de base para cargar todos los componentes del applet, esta clase es la que tiene todos los métodos que sirven para el funcionamiento intrínseco del proyecto. Sobre ella se apoyan además dos clases internas que son las que pintarán los resultados (*ZonaGrafica*) y otra que manejará los tres hilos de ejecución cuando la representación sea dinámica.

Esta clase da al usuario las herramientas para poder interactuar con el applet. Además de todas las cajas de texto que deberán ser rellenadas para poder recoger los valores iniciales y los parámetros, se dan herramientas de control de gráfica, cambio de idioma, posibilidad de poner cuadrícula o numeras los ejes, cambiar velocidad, imprimir y guardar imagen generada. Todo esto es llevado a cabo gracias sobre todo a "listeners" que actúan como métodos que escuchan los eventos que suceden dentro de este panel y reaccionan en consecuencia, por ejemplo, el pulsado de un botón.

Clase Punto:

La clase *Punto* nos da la funcionalidad de crear objetos con dos atributos, la coordenada *x* y la coordenada *y*. Es una clase muy útil para la representación ya que lo que pintamos en el dibujo son un gran número de puntos. Contiene los métodos típicos de *get* y *set* para poder acceder a ellos.

Clase Orbita:

Esta clase nos da la posibilidad de crear objetos de tipo *Orbita* que dependiendo de cuál sea la ecuación, sus componentes y por tanto sus *get* y *set* variarán en tamaño.

Como elementos comunes entre los distintos objetos *Orbita* que construiremos, tenemos los atributos *t*, *x* e *y* que estarán en cada uno de ellos. Para dar una mejor funcionalidad a este fenómeno, hemos decidido crear esta clase como abstracta y que contenga todos estos elementos comunes entres distintas ecuaciones.

En esta clase por tanto además de los atributos ya dichos tendremos los métodos comunes: *get* y *set* para todos los atributos dichos además de uno para un objeto Punto y otro para una lista de objetos Punto que contendrán los resultados que más tarde mostraremos por pantalla.

Clase OrbEcuacionN:

Aunque esta clase no viene en el diagrama de clase la explicaremos aquí. En esta clase se hará la distinción para cada ecuación. Cada una de las clases *OrbEcuacionN* heredará de la clase *Orbita*. Por tanto todos los atributos y métodos, también podrán ser usados desde esta clase.

Además de los métodos típicos de *get* y *set* para los distintos atributos que referencian a los parámetros propios de cada ecuación, tendremos otros: El constructor que forma la órbita entera de cada ecuación y el más importante (*calculaTrayectoria()*) que carga sobre una lista de puntos el resultado de llamar al método *resolver()* de la clase *Rungekutta* que veremos a continuación.

Clase RungekuttaN:

Esta clase es esencial en nuestro proyecto, ya que se ocupa de realizar los cálculos sobre la ecuación correspondiente para poder dar la lista de puntos a pintar. Para ello realiza una integración numérica en cuatro pasos.

Clase ZonaGrafica:

Es una clase interna de *IUEcuacionN* que dará la funcionalidad al proyecto para pintar sobre el panel correspondiente, todos los resultados que necesitemos.

Clase Hilo:

Al igual que la anterior, esta clase también será interna de *IUEcuacionN.* Sin embargo, esta clase servirá para el control de ejecución en modo dinámico de la representación.

En esta clase, tendremos un único atributo que hará cargar un objeto de clase *Hilo* y sobre él podremos simular la dinámica de representación. Podremos, además de iniciarlo, pausarlo o pararlo según el usuario quiera con los controles que se le dan en la interfaz.

3.2.5. Javadoc:

Gracias al IDE de NetBeans o a Eclipse podemos tener una base de documentos Java para formalizar nuestras clases. Esta utilidad llamada Javadoc de Sun Microsystems genera documentación de API's en formato HTML a partir del código. A continuación, enseñaremos una muestra de esta utilidad con la clase *Orbita:*

graficos

Class Orbita

```
java.lang.Object
  └graficos.Orbita
```

```
public abstract class Orbita
extends java.lang.Object
```

Constructor Summary
Orbita()
Orbita(double x, double y, double t)

Method Summary	
java.util.List<graficos.Punto>	getArrayPuntos()
graficos.Punto	getP_inicial()
double	getT()
double	getX()
double	getY()
void	setArrayPuntos(java.util.List<graficos.Punto> arrayPuntos)
void	setP_inicial(graficos.Punto p_inicial)
void	setT(double t)
void	setX(double x)
void	setY(double y)

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Orbita

```
public Orbita()
```

Orbita

```
public Orbita(double x,
              double y,
              double t)
```

Method Detail

getX

```
public double getX()
```

setX

```
public void setX(double x)
```

getY

```
public double getY()
```

setY

```
public void setY(double y)
```

getT

```
public double getT()
```

setT

```
public void setT(double t)
```

setP_inicial

```
public void setP_inicial(graficos.Punto p_inicial)
```

getP_inicial

```
public graficos.Punto getP_inicial()
```

getArrayPuntos

```
public java.util.List<graficos.Punto> getArrayPuntos()
```

setArrayPuntos

```
public void setArrayPuntos(java.util.List<graficos.Punto> arrayPuntos)
```

3.3. Modo de uso

Una vez arrancado el applet, lo primero que nos aparece es la ecuación del *Oscilador de Helmholtz*. En ella nos aparecen en las cajas de texto unos valores por defecto. Esta primera representación y para todas cuando cambiemos de sistema, es estática. Por tanto, el sistema está listo para poder representar la órbita 1. Aún así, podemos antes de ejecutar la onda, cambiar el color o el grosor de ésta. Ahora ya podemos dar al botón de *Empezar* para representar la primera onda del sistema del *Oscilador de Helmholtz*. Se puede ver en la barra de progreso de debajo del panel de representación, que se pone al 100% nada más pulsar el botón de comienzo, ya que al ser estático, es inmediato.

Una vez realizada esta primera representación, podemos tomar varios caminos. Uno de ellos y el más sencillo sería elegir la pestaña 2 de órbita (*O2*) y la siguiente con sus consecuentes parámetros, color y grosor propios. Cabe notar que el sistema está definido para que la primera órbita a representar sea la primera (*O1*) que nos sirve para definir los valores máximos de los puntos a representar y se vea toda la gráfica dentro del panel de representación. Si quisiéramos comenzar la representación de gráficas con la segunda o la tercera órbita, nos saldría un mensaje de advertencia que nos diría que primero rellenáramos los datos (parámetros) de la primera órbita.

La siguiente opción que podríamos tomar sobre esta ventana, sería la de cambiar la representación a dinámica que sería tan fácil como pulsar el botón *Representación estática.* Una vez hecho esto, los parámetros y los valores iniciales se resetean y quedan vacíos. Una vez rellenados con valores, podemos dar al botón de *Empezar* con la misma restricción de tener que empezar con la primera onda. Una vez hecho esto y gracias a los controles que nos da la clase *Hilo* (Thread), podemos pausar la ejecución con el mismo botón que nos permitía empezar la ejecución, o podemos pararla y resetear los valores con el botón *Resetear.* También y como elementos exclusivos de la ejecución dinámica, se da uso a dos componentes: botón de *tijeras* (permite mientras se está ejecutando una onda activa, no pausada, borrar el dibujo de la gráfica hecho justo antes de haber pulsado este botón) y barra de velocidad (barra deslizante que permite realizar la representación más o menos rápido según el punto en el que este el marcador).

Hay que notar que en los dos modos de representación, tanto estático como dinámico, el *applet* nos permite poder desplazarnos sobre las gráficas y realizar zoom de alejamiento o acercamiento. Además, en los dos modos también podemos elegir exclusivamente entre ellos, con sendos botones (*Con cuadrícula* o *Ejes numerados*), si ponemos cuadrícula o ejes numerados a la representación.

Por último, dentro de este panel de ecuación tenemos 3 botones secundarios que dan más valor al applet. Uno que permite el cambio de idioma, de español a inglés y viceversa, otro botón que nos permite guardar la imagen generada en el panel de representación y el último que da la opción de imprimir esta misma imagen.

La otra opción que podemos tomar, una vez hecho todo lo descrito, es el cambio de ecuación. Esta opción se da al usuario mediante un menú desplegable que permite un a clara muestra de los cinco tipos de sistemas a visualizar.

3.4. Implementación

En esta parte nos pararemos a detallar el código de las clases y métodos que conforman el *applet*. Obviamente deberemos realizar un pequeño resumen de ello, centrándonos sobre las partes más esenciales y sobre aquellas que presenten cierta dificultad para su entendimiento.

Clase IvofApplet:

Esta clase sirve de soporte a todo el proyecto ya que es una extensión de la clase *JApplet*:

```
public class IvofApplet extends javax.swing.JApplet {
```

Sobre está clase se cargarán los distintos paneles que darán cabida a las ecuaciones de los sistemas a visualizar:

```
private IUEcuacion1 iUEcuacion1 = new IUEcuacion1(this);
private IUEcuacion2 iUEcuacion2 = new IUEcuacion2(this);
private IUEcuacion3 iUEcuacion3 = new IUEcuacion3(this);
private IUEcuacion4 iUEcuacion4 = new IUEcuacion4(this);
private IUEcuacion5 iUEcuacion5 = new IUEcuacion5(this);
```

Además de estos paneles, la clase IvofApplet cargará dos componentes más que tendrán sus respectivos "listeners" para que cualquier acción del usuario tenga la reacción debida.

El primero es un menú desplegable llamado eleccionActionPermorfed (ActionEvent evt) que nos permitirá elegir la ecuación que queremos visualizar:

```
private void eleccionActionPerformed(ActionEvent evt) {
	String cad1=(String)eleccion.getSelectedItem();
	panelPrincipal.removeAll();
	panelPrincipal.add(BorderLayout.NORTH,eleccion);
	panelPrincipal.add(BorderLayout.SOUTH,enlaceWeb);
	if (cad1.equals("Oscilador de Helmholtz")) {
		panelPrincipal.add(BorderLayout.CENTER,iUEcuacion1);
	}
	else if (cad1.equals("Péndulo simple(v. angular)")){
		panelPrincipal.add(BorderLayout.CENTER,iUEcuacion2);
	}
	else if (cad1.equals("Péndulo doble(x,y)")){
		panelPrincipal.add(BorderLayout.CENTER,iUEcuacion3);
	}
```

```
        else if (cad1.equals("Péndulo doble(v. angular)")){
                panelPrincipal.add(BorderLayout.CENTER,iUEcuacion4);
        }
        else if (cad1.equals("Péndulo simple(x,y)")){
                panelPrincipal.add(BorderLayout.CENTER,iUEcuacion5);
        }
        panelPrincipal.updateUI();
        panelPrincipal.repaint();
}
```

El segundo es una *JLabel* o etiqueta que convertiremos en una enlace a la página web del Departamento de Física de la URJC:

```
enlaceWeb.addMouseListener(new java.awt.event.MouseAdapter() {
        public void mouseClicked(java.awt.event.MouseEvent evt) {
                enlaceWebMouseClicked(evt);
        }
});
```

Todos estos elementos serán cargados por un método al que llamaremos desde el método inicial *init()*. Este método se encargará de colocar sobre el panel todos estos elementos ya descritos que conforman el *applet*.

```
protected void jbInit() {
        panelPrincipal=new JPanel();
        panelPrincipal.setLayout(new BorderLayout());
        panelPrincipal.add(BorderLayout.NORTH,eleccion);
        panelPrincipal.add(BorderLayout.CENTER,iUEcuacion1);
        panelPrincipal.add(BorderLayout.SOUTH,enlaceWeb);
        add(panelPrincipal);
}
```

Aquí podemos ver, como se cargá el panel *panelPrincipal* que dará cabida a todos los componentes, entre los que están: menú desplegable *eleccion* (arriba), panel de la primera ecuación u *Oscilador de Helmholtz* (centro) y el enlace al Departamento de Física de la URJC (abajo).

Clase IUEcuacionN:

Aunque la anterior clase es la extensión de la clase *JApplet*, esta es la más importante ya que es dónde se darán todos aquellos métodos que harán de este *applet* una herramienta para el usuario y dónde él podrá interactuar con él.

En primer lugar y como acabamos de decir, mostraremos el primero de los métodos esenciales de este panel. Será aquel que recoja todos los valores iniciales y parámetros que el usuario decida introducir al sistema y el comportamiento del sistema ante cualquier tipo de anomalía, como por ejemplo la introducción de caracteres alfabéticos.

Es importante recalcar que no habrá casi diferencias entre las clases de IUEcuacionN. Aún así intentaremos explicar en que momento y porque se dan éstas.

```
public void leerParametros(Orbita orb, int nOrbita){
	String x,y,t;
	String m = null;
	String b = null;
	String k = null;
	String f = null;
	String w = null;
	double mNum = 0.0;
	double bNum = 0.0;
	double kNum = 0.0;
	double fNum = 0.0;
	double wNum = 0.0;
	double xNum = 0.0;
	double yNum = 0.0;
	double tNum = 0.0;

	if (nOrbita == 1){
		m = parametro1_1_1.getText();
		b = parametro1_1_2.getText();
		k = parametro1_1_3.getText();
		f = parametro1_1_4.getText();
		w = parametro1_1_5.getText();
	}
	else if (nOrbita == 2){
		m = parametro1_2_1.getText();
		b = parametro1_2_2.getText();
		k = parametro1_2_3.getText();
		f = parametro1_2_4.getText();
		w = parametro1_2_5.getText();
	}
	else if (nOrbita == 3){
		m = parametro1_3_1.getText();
		b = parametro1_3_2.getText();
		k = parametro1_3_3.getText();
		f = parametro1_3_4.getText();
		w = parametro1_3_5.getText();
	}

	x = varX1.getText();
	y = varY1.getText();
	t = varT1.getText();

	xNum = Double.parseDouble(x);
	yNum = Double.parseDouble(y);
	tNum = Double.parseDouble(t);

	if (!("".equals(m))){
		mNum = Double.parseDouble(m);
	}
	if (!("".equals(b))){
		bNum = Double.parseDouble(b);
	}
```

--- similar para todos los parámetros

```
	if (nOrbita == 1)
	{
	 o1 = new OrbEcuacion1(xNum,yNum,tNum,mNum,bNum,kNum,fNum,wNum);
	}
```

```
        else if (nOrbita == 2)
        {
         o2 = new OrbEcuacion1(xNum,yNum,tNum,mNum,bNum,kNum,fNum,wNum);
              }
        else if (nOrbita == 3){
         o3 = new OrbEcuacion1(xNum,yNum,tNum,mNum,bNum,kNum,fNum,wNum);
        }
}
```

Este método recoge todos los parámetros introducidos en los *JTextField* y los asigna según el entero introducido en la cabecera, a la onda que haya sido seleccionada. Aquí en este método no se observa, pero en la definición de los *JTextField* hemos realizado ésta ayudándonos de una clase que verifica que los parámetros no sean nulos ni caracteres alfabéticos.

--- definición de los *JTextField*

```
parametro1_3_1.setInputVerifier(new Verificador());
```

--- clase Verificador

```
public class Verificador extends InputVerifier {
    public boolean verify(JComponent editor) {
        if (editor instanceof JTextField){
            String clave = ((JTextField)editor).getText();
            try{
                Double.parseDouble(clave);
                return true;
            }
            catch (Exception e){
                ((JTextField)editor).setText("");
                return false;
            }
        }
        return true;
    }
}
```

La clase *ZonaGrafica* se encarga de la representación de los resultados (puntos o péndulo) obtenidos según los parámetros recogidos por el método anterior. Además de esto, también dibuja los ejes y si el usuario lo considera oportuno la cuadrícula o la numeración de los ejes.

Por último interpreta las pulsaciones en los botones de desplazamiento y zoom que el usuario realice para adaptarlo a la visualización de las gráficas.

```
class ZonaGrafica extends JPanel{
      public void paintComponent(Graphics g){
            super.paintComponent(g);
            Graficar(g);     //x0,y0 se inicalizan en init
      }

      public void Graficar(Graphics ap){
            Graphics2D g = (Graphics2D) ap;
            g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
            double Gancho,Galto;
```

```
double mitadx,mitady;
Gancho = getSize().width;
Galto = getSize().height;
mitadx = Gancho/2;
mitady = Galto/2;
g.setPaint(Color.BLACK);
g.setStroke(grosor);
//eje Y
g.draw(new Line2D.Double(mitadx-ejeX, 0,mitadx-ejeX,
Galto));
//eje X
g.draw(new Line2D.Double(0, mitady+ejeY, Gancho,
mitady+ejeY));
if ((!est_din)&&(play1pulsado)&&(!stopPulsado)){//dinamico
   int nPuntosLocal1,nPuntosLocal2,nPuntosLocal3;
   if (final1){
      double x1,y1,x2,y2;
      maxX = MaximoX(listaPuntos1);
      maxY = MaximoY(listaPuntos1);
      mostrarProgreso(listaPuntos1.size(),1);
      for (int i=pTijeras1; i<listaPuntos1.size()-1;i++){
         x1 = Escala(mitadx,listaPuntos1.get(i).getX(),
              maxX);
         y1 = Escala(mitady,listaPuntos1.get(i).getY(),
              maxY);
         x2 = Escala(mitadx,listaPuntos1.get(i+1).getX(),
              maxX);
         y2 = Escala(mitady,listaPuntos1.get(i+1).getY(),
              maxY);
         g.setColor(ecolor1);
         g.setStroke(grosor1);

         if (i==pTijeras1){

            PintaInicio(g,x1,mitadx,mitady,y1);
         }
         if (i==(listaPuntos1.size()-2)){
            PintaFinal(g,x2,mitadx,mitady,y2);
         }
         g.draw(new Line2D.Double((zoom*x1)+mitadx-
         ejeX,mitady-(y1*zoom)+ejeY,(zoom*x2)+mitadx-
         ejeX,mitady-(y2*zoom)+ejeY) );
      }
   }

   nPuntosLocal1 = nPuntos1;
   if ((listaPuntos1.size()-1)>nPuntosLocal1){
      double x1,y1,x2,y2;
      maxX = MaximoX(listaPuntos1);
      maxY = MaximoY(listaPuntos1);
      for (int i=pTijeras1; i<nPuntosLocal1;i++) {
         x1 = Escala(mitadx,listaPuntos1.get(i).getX(),
              maxX);
         y1 = Escala(mitady,listaPuntos1.get(i).getY(),
              maxY);
         x2 = Escala(mitadx,listaPuntos1.get(i+1).getX(),
              maxX);
         y2 = Escala(mitady,listaPuntos1.get(i+1).getY(),
              maxY);
```

```
                g.setColor(ecolor1);
                g.setStroke(grosor1);
                if (i==pTijeras1){
                   PintaInicio(g,x1,mitadx,mitady,y1);
                }
                g.draw(new Line2D.Double((zoom*x1)+mitadx-
                ejeX,mitady-(y1*zoom)+ejeY,(zoom*x2)+mitadx-
                ejeX,mitady-(y2*zoom)+ejeY));
             }
             x1 = Escala(mitadx,listaPuntos1.get
                 (nPuntosLocal1).getX(), maxX);
             y1 = Escala(mitady,listaPuntos1.get
                 (nPuntosLocal1).getY(),maxY);
             x2 = Escala(mitadx,listaPuntos1.get
                 (nPuntosLocal1+1).getX(),maxX);
             y2 = Escala(mitady,listaPuntos1.get
                 (nPuntosLocal1+1).getY(),maxY);
             mostrarProgreso(nPuntosLocal1,1);
             g.setColor(ecolor1);
             g.setStroke(grosor1);
             PintaFinal(g,x2,mitadx,mitady,y2);
             g.draw(new Line2D.Double((zoom*x1)+mitadx-
             ejeX,mitady-(y1*zoom)+ejeY,(zoom*x2)+mitadx-e
             jeX,mitady-(y2*zoom)+ejeY));
          }

          if (final2){
             --- lo mismo para la órbita 2
          }
          if (final3){
             --- lo mismo para la órbita 3
          }
       }//final dinamico

       else if (est_din){ //estatico
          if (!listaPuntos1.isEmpty()) {
             double x1,y1,x2,y2;
             maxX = MaximoX(listaPuntos1);
             maxY = MaximoY(listaPuntos1);
             for (int i=0; i<listaPuntos1.size()-1;i++){
                x1 = Escala(mitadx,listaPuntos1.get(i).getX(),
                     maxX);
                y1 = Escala(mitady,listaPuntos1.get(i).getY(),
                     maxY);
                x2 = Escala(mitadx,listaPuntos1.get(i+1).getX(),
                     maxX);
                y2 = Escala(mitady,listaPuntos1.get(i+1).getY(),
                     maxY);
                g.setColor(ecolor1);
                g.setStroke(grosor1);
                if (i==0){
                   PintaInicio(g,x1,mitadx,mitady,y1);
                }
                if(i==(listaPuntos1.size()-2)){
                   PintaFinal(g,x2,mitadx,mitady,y2);
                }
                g.draw(new Line2D.Double((zoom*x1)+mitadx-
                ejeX,mitady-(y1*zoom)+ejeY,(zoom*x2)+mitadx-
```

```
            ejeX,mitady-(y2*zoom)+ejeY) );
        }
    }

    if (!listaPuntos2.isEmpty()) {
```
--- lo mismo para la órbita 2
```
    }
    if (!listaPuntos3.isEmpty()) { //tercera onda
```
--- lo mismo para la órbita 3
```
    }
}//final estatico
```

Seguidamente veremos el proceso para poner la cuadrícula adaptada para cualquier visualización:

```
if (cuadricula){/
    int iczoom;
    int cxmin,cxmax,cymin,cymax;
    double xmin,xmax;
    int iejeXIzq=0;
    int iejeXDer=0;
    int iejeYArr=0;
    int iejeYAba=0;

    if (listaPuntos1.isEmpty()){
        xmin=-1.0*mitadx/escalaX;
        xmax=(1.0*(Gancho-mitadx)/escalaX);
        cxmin=(int)Math.round(xmin);
        cxmax=(int)Math.round(xmax);
        cymin=(int)Math.round(1.0*(mitady-Galto)/escalaY);
        cymax=(int)Math.round(1.0*mitady/escalaY);
        g.setPaint(Color.GRAY);
        for (int i=cxmin;i<cxmax+1;i++){
            g.draw(new Line2D.Double(mitadx+i*escalaX, 0,
            mitadx+i*escalaX , Galto));
        }
        for (int i=cymin;i<cymax+1;i++){
            g.draw(new Line2D.Double(0, mitady-i*escalaY,
            Gancho , mitady-i*escalaY));
        }
    }
    else{
        cxmin=-1*(int)Math.floor(maxX);
        cxmax=(int)Math.floor(maxX);
        cymin=-1*(int)Math.floor(maxY);
        cymax=(int)Math.floor(maxY);
        g.setPaint(Color.GRAY);

        if (zoom>=1){
            iczoom=1;
        }
        else{
            iczoom=(int)Math.round(1/zoom);
        }
        if (iejeX<0){
            iejeXIzq=iejeX;
        }
```

```
            else if (iejeX>0){
               iejeXDer=iejeX;
            }
            if (iejeY<0){
               iejeYAba=iejeY;
            }
            else if (iejeY>0){
               iejeYArr=iejeY;//estaba antes la X
            }
            for (int i=(cxmin*iczoom)+iejeXIzq;
            i<(cxmax*iczoom)+1+iejeXDer;i++){//verticales
               double cix;
               cix=Escala(mitadx,i,maxX);
               g.draw(new Line2D.Double((zoom*cix)+mitadx-ejeX,
               0, (zoom*cix)+mitadx-ejeX , Galto));
            }
            for (int i=(cymin*iczoom)+iejeYAba;
            i<=(cxmax*iczoom)+1+iejeYArr;i++){ //horizontales
               double ciy;
               ciy=Escala(mitady,i,maxY);
               g.draw(new Line2D.Double(0, mitady-(ciy*zoom)
               +ejeY, Gancho , mitady-(ciy*zoom)+ejeY));
            }
         }

      }
```

Se puede ver en el primer *if* que se puede colocar la cuadrícula aunque no haya aún parámetros introducidos en el sistema. Sería un caso por defecto. En cuanto se pasa a la condición *else*, se puede observar que los máximos valores de la lista de puntos resultado dan la base para realizar la cuadrícula. Además, claro está, se tiene en cuenta si hay desplazamientos o zooms sobre la gráfica.

Ahora veremos el caso en el que se decidan poner los ejes numerados, que al igual que el caso anterior de la cuadrícula, también usará los valores máximos de los resultados y tendrá en cuenta los mismos aspectos.

```
      else{// sino hay cuadricula puede numerar los ejes
         if (numerar){
            int cxmin,cxmax,cymin,cymax;
            double xmin,xmax;
            xmin=-1.0*mitadx/escalaX;
            xmax=(1.0*(Gancho-mitadx)/escalaX);
            cxmin=(int)Math.round(xmin); //pantalla
            cxmax=(int)Math.round(xmax);
            cymin=(int)Math.round(1.0*(mitady-Galto)/escalaY);
            cymax=(int)Math.round(1.0*mitady/escalaY);
            g.setPaint(Color.GRAY);

            if (listaPuntos1.isEmpty()){
               for (int i=cxmin;i<cxmax+1;i++){
               float arg1 = Double.valueOf(mitadx+i*escalaX+3)
                            .floatValue();
               float arg2 = Double.valueOf(mitady-3)
                            .floatValue();
               g.drawString(""+i, arg1, arg2 );
            }
```

```
            for (int i=cymin;i<cymax+1;i++){
                float arg1 = Double.valueOf(mitadx+3)
                             .floatValue();
                float arg2 = Double.valueOf(mitady-i*escalaY-3)
                             .floatValue();
                g.drawString(""+i, arg1 , arg2 );
            }
        }
        else{
            int iczoom;
            int iejeXIzq=0;
            int iejeXDer=0;
            int iejeYArr=0;
            int iejeYAba=0;

            if (zoom>=1){
                iczoom=1;
            }
            else{
                iczoom=(int)Math.round(1/zoom);
            }
            if (iejeX<0){
                iejeXIzq=iejeX;
            }
            else if (iejeX>0){
                iejeXDer=iejeX;
            }
            if (iejeY<0){
                iejeYAba=iejeY;
            }
            else if (iejeY>0){
                iejeYArr=iejeY;//estaba antes la X
            }
            g.setPaint(Color.GRAY);
            for (int i=(cxmin*iczoom)+iejeXIzq;i<(cxmax*iczoom)
            +1+iejeXDer;i++){ //verticales
                double cix;
                cix=Escala(mitadx,i,maxX);
                float arg1 = Double.valueOf((zoom*cix)+mitadx-
                             ejeX+3).floatValue();
                float arg2 = Double.valueOf(mitady+ejeY-3)
                             .floatValue();
                g.drawString(""+i, arg1, arg2 );
            }
            for (int i=(cymin*iczoom)+iejeYAba;i<=(cxmax*iczoom)
            +1+iejeYArr;i++){ //horizontales
                double ciy;
                ciy=Escala(mitady,i,maxY);
                float arg1 = Double.valueOf(mitady-(ciy*zoom)
                             +ejeY-3).floatValue();
                float arg2 = Double.valueOf(mitady-ejeX+58)
                             .floatValue();
                g.drawString(""+i, arg2 , arg1 );
            }
        }
```

Además dentro de esta clase interna de *ZonaGrafica* tenemos otros métodos que variarán según que ecuación tengamos delante. Esto se hará patente entre las

ecuaciones que usen péndulos en su modo de X frente a Y y durante la representación dinámica. En el resto de casos se hará muy parecido.

```
public void PintaInicio(Graphics2D g,double x,double mitadx,double
mitady,double y){
      Ellipse2D.Double cInicio = new Ellipse2D.Double((zoom*x)+mitadx-
                                 ejeX,mitady-(y*zoom)+ejeY, 8, 8);
      g.fill(cInicio);
}

public void PintaFinal(Graphics2D g,double x,double mitadx,double
mitady,double y){
      Ellipse2D.Double cFinal = new Ellipse2D.Double((zoom*x)+mitadx-
                                 ejeX,mitady-(y*zoom)+ejeY, 8, 8);
      g.draw(cFinal);
}
```

Estos dos métodos se encargan de pintar al principio de cada gráfica un círculo relleno y al final un círculo sin relleno para controlar dónde comenzó y dónde terminó la representación.

En el caso del péndulo doble y del péndulo simple donde se representa la X frente a la Y en el modo dinámico, se visualiza un péndulo tal cual, por tanto, se deberán ofrecer otros métodos. Podremos ver este caso para el péndulo doble dónde además veremos como según el valor de la masa, el tamaño del péndulo varía.

```
public void PintaFinal(Graphics2D g,double x,double mitadx,double
mitady,double y, double m, double o){
   if (m == 1){//masa 1
      double m1;
      if (o == 1){
         m1 = o1.getM1();
         Ellipse2D.Double cFinal = new Ellipse2D.Double((zoom*x)
                                   +mitadx-ejeX,mitady-(y*zoom)+ejeY,
                                   8+m1, 8+m1);
         if (est_din){//estatico
            g.draw(cFinal);
         }
         else{//dinamico
            g.fill(cFinal);
         }
      }
      else if (o == 2){
         --- lo mismo para la órbita 2
      }
      else if (o == 3){
         --- lo mismo para la órbita 3
      }
   }
   else if (m == 2){//masa 2
      --- lo mismo para la masa 2
   }
```

```
public void PintaPendulo(Graphics2D g,double mitadx,double
mitady,double maxX2,double x2_1,double y2_1,double x2_2,double y2_2){
   double x0,y0;
   x0 = Escala(mitadx,0.0,maxX2);
   y0 = Escala(mitadx,0.0,maxX2);
   g.setColor(Color.BLACK);
   g.setStroke(new BasicStroke(2.5f));
   g.draw(new Line2D.Double((zoom*x0)+mitadx-ejeX,mitady-(y0*zoom)
   +ejeY,(zoom*x2_1)+mitadx-ejeX,mitady-(y2_1*zoom)+ejeY));
   g.draw(new Line2D.Double((zoom*x2_1)+mitadx-ejeX,mitady-(y2_1*zoom)
   +ejeY,(zoom*x2_2)+mitadx-ejeX,mitady-(y2_2*zoom)+ejeY));
}
```

Otra clase importante dentro de nuestra clase principal *IUEcuacionN* es la que controla la ejecución de hilos o threads. En ella se da el comportamiento necesario para simular el inicio, pausa o parada de hilos que representan cada onda en modo dinámico.

```
class Hilo extends Thread{
   int opOnda;

   public Hilo(int o){
      this.opOnda = o;
   }

   public void run(){
      while(true){
         try{
            Thread.sleep(retardo(velocidad1.getValue()));
         }catch (InterruptedException e) { }

         if (opOnda==1){// thread 1 - onda 1
            if (!pause1pulsado){
               if (stopPulsado){
                  vaciarListas();
                  break;
               }
               else if ((play1pulsado) &&
               (listaPuntos1.size()>nPuntos1)){
                  areaResultados1.repaint();
                  nPuntos1++;
               }
               if (listaPuntos1.size()==nPuntos1){
                  final1 = true;
                  areaResultados1.repaint();
                  if (esp_engl){
                      play1_1.setText("Empezar");
                  }
                  else{
                      play1_1.setText("Start");
                  }
                  break;
               }
            }
```

```
            else{
               if (stopPulsado){
                  pause1pulsado = false;
                  break;
               }
            }
         }
         if (opOnda==2){// thread 2 - onda 2
            --- lo mismo para la órbita 2
         }
         if (opOnda==3){// thread 3 - onda 3
            --- lo mismo para la órbita 3
         }
      }
}
```

Aquí podemos observar como se comportan los hilos. Una vez creado, debido a que se pulsó el botón *Empezar* de la ejecución dinámica, se va incrementando el atributo que controla el numero de puntos que se van dibujando (*nPuntos1*). Cuando el número de puntos totales de la lista resultado se iguala al contador, la ejecución se para y se sale del método. Si durante la ejecución se pulsara el botón *Pausar*, el hilo entraria en un bucle del que no saldría hasta que pulsara el botón *Reanudar*.

Hay dos posibilidades para pausar el botón de *Resetear*: cuando se está ejecutando el hilo o cuando está pausado. En cualquiera de los dos casos, la ejecución para, se borra lo dibujado y se vuelve al estado inicial.

Una vez descritas estas dos clases internas pasaremos a explicar los métodos más importantes que hay en la clase *IUEcuacionN.* Empezaremos por la que hace que se proceda a leer los parámetros e impulse el cálculo de resultados y la consiguiente visualización de ellos:

```
private void play1_1ActionPerformed(ActionEvent evt){
   if (est_din){//estatico
      leerParametros(o1,1);
      listaPuntos1 = o1.getArrayPuntos();
      areaResultados1.repaint();
      progreso1.setValue(10000);
      progreso1.setStringPainted(true);
      progreso1.setString("100%");
   }
   else{//dinamico
      String botonText = null;
      botonText = play1_1.getText();
      if ( ("Play".equals(botonText)) || ("Empezar".equals(botonText))
      ||("Reanudar".equals(botonText))||("Resume".equals(botonText))){
         if ("Play".equals(botonText)||("Resume".equals(botonText)) ){
            play1_1.setText("Pause");
         }
         if ("Empezar".equals(botonText) ||
         ("Reanudar".equals(botonText))){
            play1_1.setText("Pausa");
         }
```

```
         if (!play1pulsado){
            play1pulsado = true;
            stopPulsado = false;
            nPuntos1 = 0;
            leerParametros(o1,1);
            listaPuntos1 = o1.getArrayPuntos();
            actualizarSumaPuntos(listaPuntos1.size());
            Hilo thO1 = new Hilo(1);
            thO1.start();
         }
         else{ // se reanuda el dibujo de puntos
            pause1pulsado = false;
         }
      }
      else if (("Pause".equals(botonText)) ||
      ("Pausa".equals(botonText))){
         if ("Pausa".equals(botonText)){
            play1_1.setText("Reanudar");
         }
         if ("Pause".equals(botonText)){
            play1_1.setText("Resume");
         }
         pause1pulsado = true;
      }
   }
}
```

Primero dentro del método tenemos la distinción de si la representación es estática o dinámica.

En el primer caso, el código es sencillo. Se leen los parámetros y se asignan a la onda elegida. Seguidamente se pintan los puntos resultado de los cálculos de *Rungekutta* y luego se rellena al 100% la barra de progreso.

En el segundo caso, se hace una distinción entre si el botón ha sido pulsado para empezar una ejecución, para pausarla o para reanudarla. En el caso de que se vaya a empezar, el código es similar al de la representación estática donde se leen los parámetros y se asignan a una órbita. Además se crea el objeto *Hilo* para que pueda ser controlada por su clase y se procede a controlar el número de puntos totales para mostrar el progreso de la ejecución.

Como este método procede a dar comienzo la ejecución del sistema, tendremos que tener uno que termine y resetee el sistema para futuros cálculos.

```
private void stop1_1ActionPerformed(java.awt.event.ActionEvent evt){
   if (est_din){
      JOptionPane jop = new JOptionPane();
      jop.showMessageDialog(areaIO1, "No hay ejecución dinámica",
      "Error", JOptionPane.ERROR_MESSAGE);
   }
   if (play1pulsado){
      stopPulsado = true;
      restaurarAtributos();
      vaciarListas();
      areaResultados1.repaint();
   }
}
```

Ahora llevaremos a cabo la descripción de otros métodos dentro de esta clase menos importantes que los anteriores, pero que dan mucha funcionalidad a herramientas que tenemos dentro del *applet*.

Empezaremos por los botones que dan color a la gráfica. Es un código que asigna al atributo *ecolor1* su correspondiente color para esa onda en concreto:

```
private void colorAzul1_1ActionPerformed(ActionEvent evt){
   if (est_din){
      ecolor1 = Color.BLUE;
   }
   else{
      if (!play1pulsado){
            ecolor1 = Color.BLUE;
      }
   }
}
```

El siguiente se encargará de capturar el momento exacto en el que se pulse el botón *tijeras* sólo cuando se realice una representación dinámica:

```
private void tijeras1_2ActionPerformed(ActionEvent evt){
   if ( (play2pulsado) && (!tijeras2) ){
      tijeras2 = true;
      pTijeras2 = nPuntos2;
   }
}
```

Para llevar a cabo el zoom-in y el zoom-out dentro de nuestras gráficas, se echa mano al igual que en los casos anteriores a métodos de *ActionPerformed.* En este caso, estos métodos a su vez llamarán a otros que de verdad serán los que controlen este proceso:

```
public void hacerZoomIn(){
   if ( (zoom!=32) && (!listaPuntos1.isEmpty()) ){
      zoom = zoom*2;
      areaResultados1.repaint();
   }
}

public void hacerZoomOut(){
   if ( (zoom!=0.03125) && (!listaPuntos1.isEmpty()) ){
      zoom = zoom/2;
      areaResultados1.repaint();
   }
}
```

Al igual que los anteriores, los métodos que vamos a describir son llamados por aquellos que actúan de "listeners" de los botones de desplazamiento dentro de la gráfica.

```
public void hacerDesplazamientoX(){
   if ( (iejeX<=15) && (iejeX>=-15)&& (!listaPuntos1.isEmpty()) ){
      ejeX = iejeX*30;
      areaResultados1.repaint();
   }
}

public void hacerDesplazamientoY(){
   if ( (iejeY<=15) && (iejeY>=-15)&& (!listaPuntos1.isEmpty()) ){
      ejeY = iejeY*30;
      areaResultados1.repaint();
   }
}
```

Ahora mostraremos el código de los métodos que permiten imprimir la imagen encuadrada en el panel de representación. En el primero se nos muestra la caja o diálogo que permite guardar esa imagen en la carpeta y con el nombre que deseemos.

```
private void imprimir1ActionPerformed(ActionEvent evt){
   try {
      PrinterJob job = PrinterJob.getPrinterJob();
      job.setPrintable(this);
      job.printDialog();
      job.print();
   } catch (PrinterException ex) {
      Logger.getLogger(IUEcuacion1.class.getName()).log(Level.SEVERE,
      null, ex);
   }
}
```

El segundo código de método es el que captura la imagen generada en el panel *areaResultados1* que es dónde dibujamos los resultados

```
public int print(Graphics g, PageFormat page, int p)
throws PrinterException {
   if (p > 0) {
      return NO_SUCH_PAGE;
   }
   Graphics2D g2d = (Graphics2D)g;
   g2d.translate(page.getImageableX(), page.getImageableY());
   areaResultados1.printAll(g);
   return PAGE_EXISTS;
}
```

Clase Orbita:

Esta clase es fundamental para llevar a cabo el proyecto, ya que sirve para crear objetos de tipo *Orbita* con las componentes del *Punto* y los demás parámetros asociados a una onda que introduzca el usuario.

Tenemos cinco paneles que aúnan cinco ecuaciones. Cada una de ellas tendrá a su vez tres posibles órbitas que conformar. En nuestro caso, para dar más flexibilidad a estas órbitas, hemos decidido crear esta clase como abstracta

```
public abstract class Orbita {
	…
}
```

, ya que permite aunar todos los métodos y atributos comunes a todas las órbitas. Esto significa que todos ellos (métodos y atributos) son públicos para las clases *OrbEcuacionN* que la necesite. Aún así, este procedimiento sólo hemos podido sacarle provecho para tres de las cinco ecuaciones. Para los dos casos de la ecuación del péndulo doble traía problemas, ya que no contenían en los valores iniciales una X y una Y, sino dos de ellas, ya que tenemos dos péndulos con sus correspondientes condiciones iniciales.

La clase *Orbita* abstracta se mostrará precediendo a la clase *OrbEcuacion1* que la usará sin ningún problema para aunar los métodos comunes con los de *OrEcuacion2* y *OrbEcuacion5* que corresponden al *Oscilador de Helmholtz* y al péndulo simple, respectivamente:

```
public abstract class Orbita {
	private Punto p_inicial;
	private double x,y,t;
	private List<Punto> arrayPuntos;

	public Orbita(){
		this.x = 0.0;
		this.y = 0.0;
		this.t = 0.0;
		this.setP_inicial(new Punto(getX(),getY()));
	}

	public Orbita(double x, double y, double t){
		this.x = x;
		this.y = y;
		this.t = t;
		this.setP_inicial(new Punto(getX(),getY()));
	}

	public double getX() {
		return x;
	}

	public void setX(double x){
		this.x = x;
	}
```

--- métodos *get* y *set* para los demás atributos

```
	public List<Punto> getArrayPuntos() {
		return arrayPuntos;
	}

	public void setArrayPuntos(List<Punto> arrayPuntos) {
		this.arrayPuntos = arrayPuntos;
	}
}
```

Como vemos, tenemos dos métodos constructores. Uno de ellos, serviría para el caso en el que no se introducen datos en los valores iniciales y otro en el caso de que sí se introduzcan.

Ahora veremos el caso de la clase *OrbEcuacion1* que extiende la clase abstracta de *Orbita.*

```
public class OrbEcuacion1 extends Orbita{
	private double m,b,k,f,w;

	public OrbEcuacion1(){
		super();
		this.m = 0;
		this.b = 0;
		this.k = 0;
		this.f = 0;
		this.w = 0;
	}
	public OrbEcuacion1(double x, double y, double t, double m,
double b, double k, double f, double w){
		super(x,y,t);
		this.m = m;
		this.b = b;
		this.k = k;
		this.f = f;
		this.w = w;
		calculaTrayectoria();
	}

	public void calculaTrayectoria() {
		setArrayPuntos(Rungekutta1.resolver(getP_inicial().getX(),
		getP_inicial().getY(), m, b, k, f, w, getT()));
	}

	public double getM() {
		return m;
	}

	public void setM(double m) {
		this.m = m;
	}
```

--- métodos *get* y *set* para los demás atributos

```
}
```

En esta clase es dónde llamamos al método *resolver* del *Rungekutta* que nos devolverá en forma de lista de puntos y pasaremos a la clase *IUEcuacionN* que corresponda.

Ahora mostraremos el código de la clase *OrbEcuacion3* que hará lo mismo que estas dos clases, pero sin poder usar la clase abstracta *Orbita*:

```
public class OrbEcuacion3{
   private Punto p_inicial1;
   private Punto p_inicial2;
   private double x1,x2,y1,y2,t,m1,l1,m2,l2;
   private List<Punto> arrayPuntos1;
   private List<Punto> arrayPuntos2;

   public OrbEcuacion3(){
      this.x1 = 0.0;
      this.y1 = 0.0;
      this.x2 = 0.0;
      this.y2 = 0.0;
      this.t = 0.0;
      this.m1 = 0.0;
      this.l1 = 0.0;
      this.m2 = 0.0;
      this.l2 = 0.0;
      this.setP_inicial1(new Punto(getX1(),getY1()));
   }

   public OrbEcuacion3(double x1, double y1, double x2, double y2,
   double t, double m1, double l1, double m2, double l2){
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
      this.t = t;
      this.m1 = m1;
      this.l1 = l1;
      this.m2 = m2;
      this.l2 = l2;
      this.setP_inicial1(new Punto(getX1(),getY1()));
      this.setP_inicial2(new Punto(getX2(),getY2()));
      calculaTrayectoria();
   }

   public void calculaTrayectoria() {
      setArrayPuntos1(Rungekutta3.resolver1(getP_inicial1().getX(),
      getP_inicial1().getY(),getP_inicial2().getX(),
      getP_inicial2().getY(), m1, l1, m2, l2, getT()));

      setArrayPuntos2(Rungekutta3.resolver2(getP_inicial1().getX(),
      getP_inicial1().getY(),getP_inicial2().getX(),
      getP_inicial2().getY(), m1, l1, m2, l2, getT()));
   }

   public double getX1() {
      return x1;
   }
```

```
    public void setX1(double x1) {
        this.x1 = x1;
    }
```

--- métodos *get* y *set* para los demás atributos

```
}
```

Como podemos ver, esta clase aúna los métodos y atributos de la clase *Orbita* con los de una clase *OrbEcuacionN* que extienda de ella. Al tener dos listas de de puntos resultado que mostrar en la representación, será indispensable tener que llamar dos veces al método *resolver* de *Rungekutta.*

Clase RungekuttaN:

Hemos tenido que diferenciar entre las clases *Orbita* según correspondieran a las ecuaciones del péndulo doble o no. En este caso que nos ocupa ahora, también deberemos.

Esta clase sirve para calcular mediante un método genérico de resolución numérica de ecuaciones diferenciales. Sólo contiene en realidad un método importante que recibirá los atributos de la órbita que ocupe en ese momento la representación. La resolución devolverá el conjunto de puntos sobre una lista.

En el caso de las ecuaciones del *Oscilador de Helmholtz* y los péndulos simples, la resolución se hará mediante cuatro pasos de integración:

```
public static List<Punto> resolver(double x, double y, double m, 
double b, double k, double F, double w, double t){
        ...
        solucion.add(0, p_inicial);

        for (int indice = 1; indice <= NIT; indice++){
                kx1 = q1(y);
                ky1 = q2(x,y,m,b,k,F,w,t_ini);
                kx2 = q1(y+H2*ky1);
                ky2 = q2((x+H2*kx1),(y + H2*ky1),m,b,k,F,w,(t_ini+H2));
                kx3 = q1(y+H2*ky2);
                ky3 = q2((x+H2*kx2),(y + H2*ky2),m,b,k,F,w,(t_ini+H2));
                kx4 = q1(y+H*ky3);
                ky4 = q2((x+H*kx3),(y + H*ky3),m,b,k,F,w,(t_ini+H));
                x = x + (H6*(kx1+2.0*kx2+2.0*kx3+kx4));
                y = y + (H6*(ky1+2.0*ky2+2.0*ky3+ky4));

                t_ini = t_ini+H;
                if (t_ini >= t){
                        break;
                }
                Punto p = new Punto(x, y);
                solucion.add(indice, p);
        }
                return solucion;
}
```

```
public static double q1(double x) {
	return x;
}

public static double q2(double x,double y,double m,double b,double
k,double F,double w,double t) {
	double resultado = 0.0;
	resultado = -(k/m)*x - (b/m)*y + (F/m)*Math.cos(w*t);
	return resultado;
}
```

Ahora veremos el caso del péndulo doble que al igual que el que acabamos de mostrar, también tiene cuatro pasos de integración. La diferencia será que habrá que dar cada paso de integración para las dos componentes de punto que tenemos, para el péndulo 1 y para el péndulo 2.

```
public static List<Punto> resolver1(double x1, double y1,double x2,
double y2, double m1, double l1, double m2, double l2, double t){
   ...
   solucion.add(0, p_inicial);
   for (int indice = 1; indice <= NIT; indice++){
      kp1 = q1(y1);
      kq1 = q1(y2);
      kr1 = q2_1(x1,y1,x2,y2,m1,l1,m2,l2,t_ini);
      ks1 = q2_2(x1,y1,x2,y2,m1,l1,m2,l2,t_ini);
      kp2 = q1(y1+H2*kr1);
      kq2 = q1(y2+H2*ks1);
      kr2 = q2_1((x1+H2*kp1),(y1+H2*kr1),(x2+H2*kq1),
             (y2+H2*ks1),m1,l1,m2,l2,(t_ini+H2));
      ks2 = q2_2((x1+H2*kp1),(y1+H2*kr1),(x2+H2*kq1),
             (y2+H2*ks1),m1,l1,m2,l2,(t_ini+H2));
      kp3 = q1(y1+H2*kr2);
      kq3 = q1(y2+H2*ks2);
      kr3 = q2_1((x1+H2*kp2),(y1+H2*kr2),(x2+H2*kq2),
             (y2+H2*ks2),m1,l1,m2,l2,(t_ini+H2));
      ks3 = q2_2((x1+H2*kp2),(y1+H2*kr2),(x2+H2*kq2),
             (y2+H2*ks2),m1,l1,m2,l2,(t_ini+H2));
      kp4 = q1(y1+H*kr3);
      kq4 = q1(y2+H*ks3);
      kr4 = q2_1((x1+H*kp3),(y1 + H*kr3),(x2+H*kq3),
             (y2+H*ks3),m1,l1,m2,l2,(t_ini+H));
      ks4 = q2_2((x1+H*kp3),(y1 + H*kr3),(x2+H*kq3),
             (y2+H*ks3),m1,l1,m2,l2,(t_ini+H));
      x1 = x1 + (H6*(kp1+2.0*kp2+2.0*kp3+kp4));
      y1 = y1 + (H6*(kr1+2.0*kr2+2.0*kr3+kr4));

      t_ini = t_ini+H;
      if (t_ini >= t){
         break;
      }
      x1aux = l1*Math.sin(x1);
      y1aux = l1*Math.cos(x1);
      Punto p = new Punto(-x1aux, -y1aux);
      solucion.add(indice, p);
   }
   return solucion;
}
```

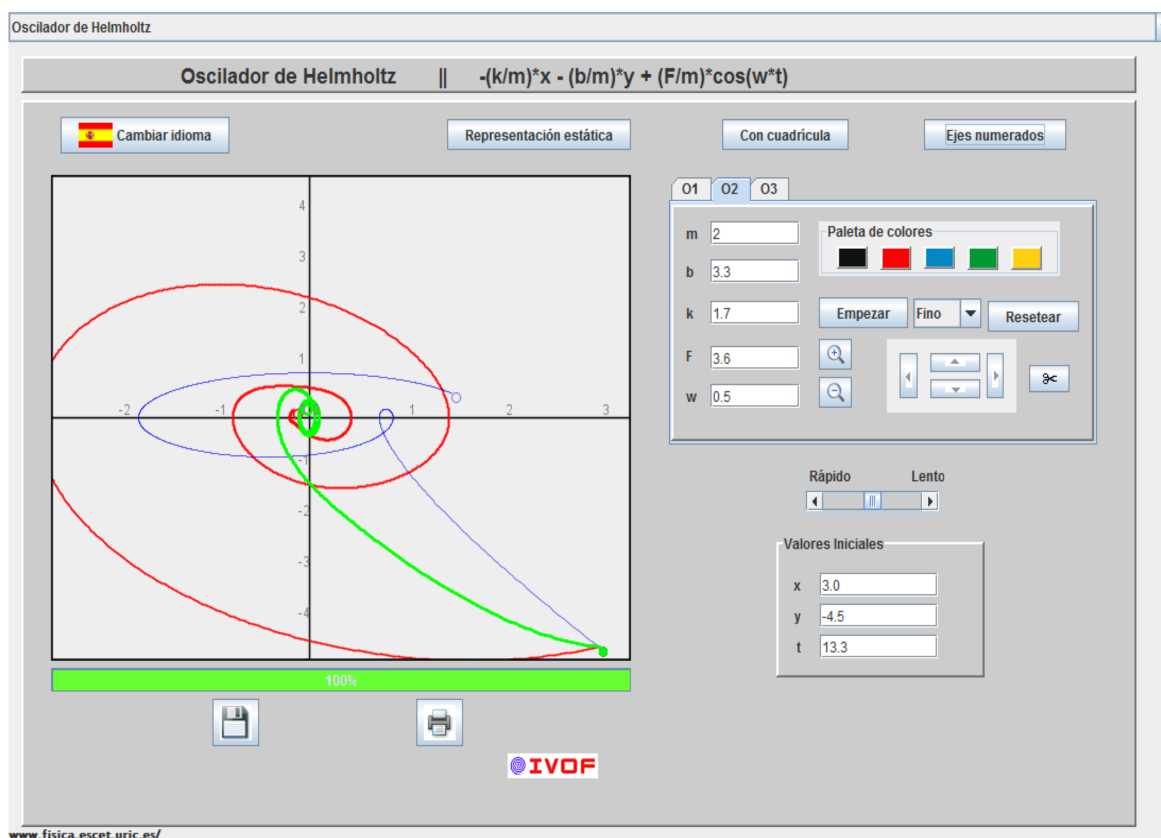
```
public static List<Punto> resolver2(double x1, double y1,double x2,
double y2, double m1, double l1, double m2, double l2, double t){

        --- similar al método anterior, pero introduciendo el valor de x2 e y2
}

public static double q1(double x) {
    return x;
}

public static double q2_1(double x1, double y1, double x2, double y2,
double m1, double l1, double m2, double l2, double t) {
    double resultado = 0.0;
    double numerador, denominador,sumando1,sumando2,mu,icosX,isenX;

    mu = 1+(m1/m2);
    icosX = Math.cos(x1-x2);
    isenX = Math.sin(x1-x2);
    sumando1 = g*( (Math.sin(x2)*icosX)-(mu*Math.sin(x1)) );
    sumando2 = ((l2*y2*y2)+(l1*y1*y1*icosX))*isenX;
    denominador = l1*(mu-(icosX*icosX));
    numerador = sumando1-sumando2;
    resultado = numerador/denominador;

    return resultado;
}

public static double q2_2(double x1, double y1, double x2, double y2,
double m1, double l1, double m2, double l2, double t) {
    double resultado = 0.0;
    double numerador, denominador,sumando1,sumando2,mu,icosX,isenX;

    mu = 1+(m1/m2);
    icosX = Math.cos(x1-x2);
    isenX = Math.sin(x1-x2);
    sumando1 = g*mu*( (Math.sin(x1)*icosX)-(Math.sin(x2)) );
    sumando2 = ( (mu*l1*y1*y1)+(l2*y2*y2*icosX))*isenX;
    denominador = l2*(mu-(icosX*icosX));
    numerador = sumando1+sumando2;
    resultado = numerador/denominador;

    return resultado;
}
```

Podemos ver claramente que los pasos de integración son los mismos, pero teniendo que hacerlo doblemente. Por esto mismo, no tenemos sólo un método *q2*, sino dos que muestran las ecuaciones de cada péndulo.

3.5. Movimientos representativos del sistema

A continuación mostraremos ciertos casos de representaciones posibles del *applet* en el que se enseñarán los cinco tipos de sistemas. Además, iremos variando entre el modo de representación estática y dinámica para hacernos una mejor idea del comportamiento de la aplicación.

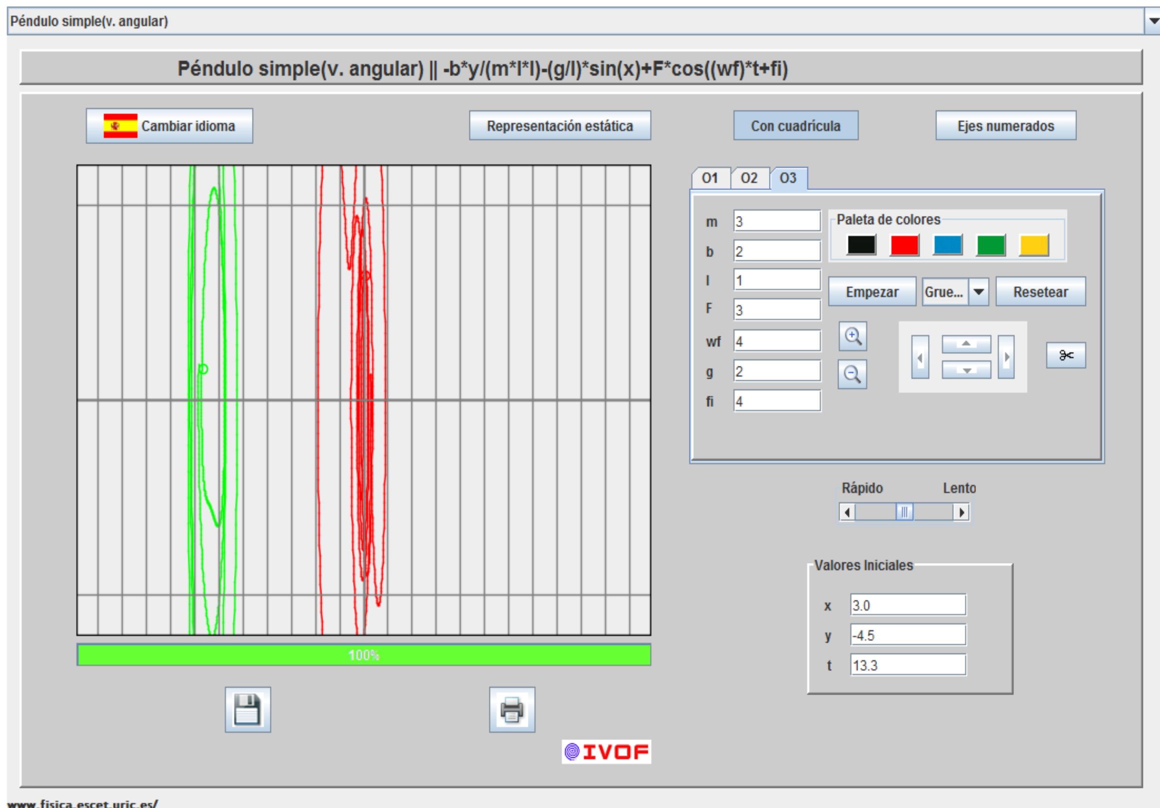
Empezaremos por las representaciones del *Oscilador de Helmholtz* con los valores iniciales y parámetros introducidos por defecto en nuestro código (*Captura 1*).



Captura 1.- Representación estática del Oscilador de Helmholtz con tres ondas

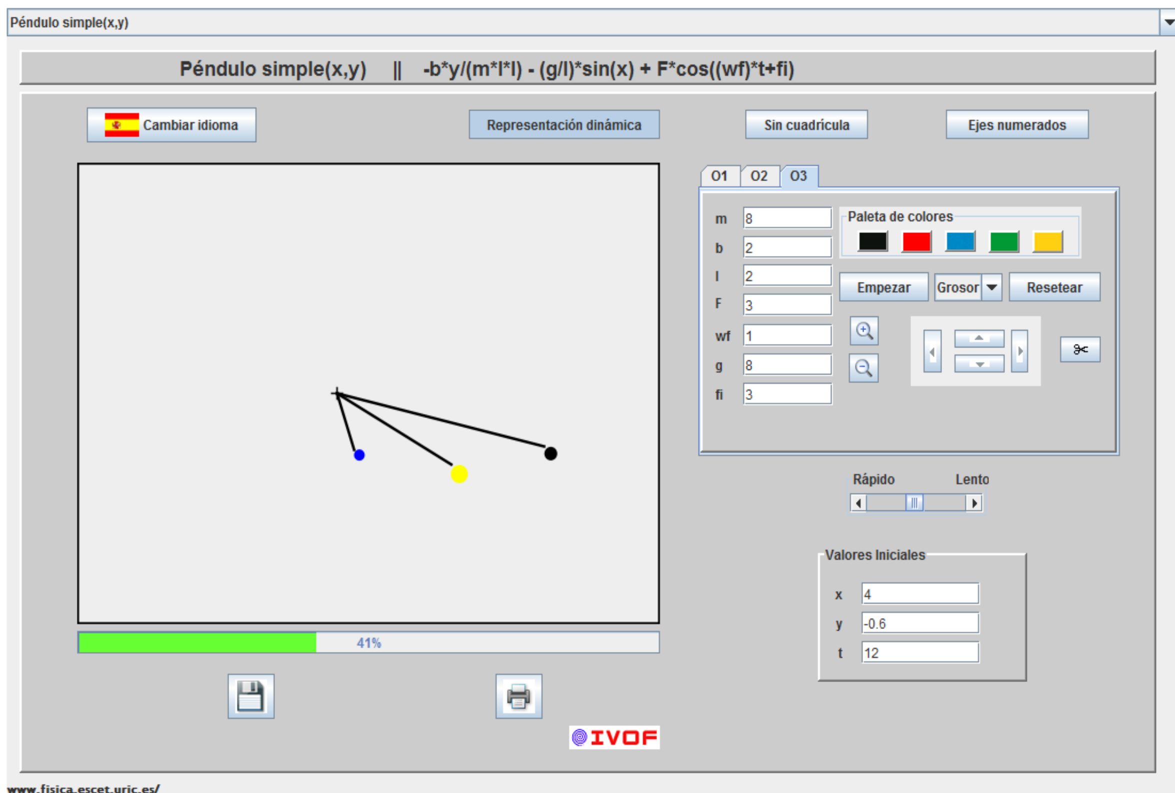
Se pueden ver claramente las 3 órbitas realizadas en modo estático, pudiendo diferenciarlas incluso en grosor. Cada una, partiendo con la misma condición inicial (3, -4.5), toman caminos distintos debido a sus parámetros. Se puede ver además los valores que van tomando gracias a la numeración de los ejes.

La siguiente representación será para mostrar el comportamiento del péndulo simple según la velocidad angular (*Captura 2*). En ella podremos ver dos órbitas (roja y verde) y su distinto comportamiento, gracias al zoom que haremos. Además hemos decidio poner una cuadrícula para medir de mejor manera su cercanía o lejanía.



Captura 2.- Representación estática del péndulo simple con dos ondas

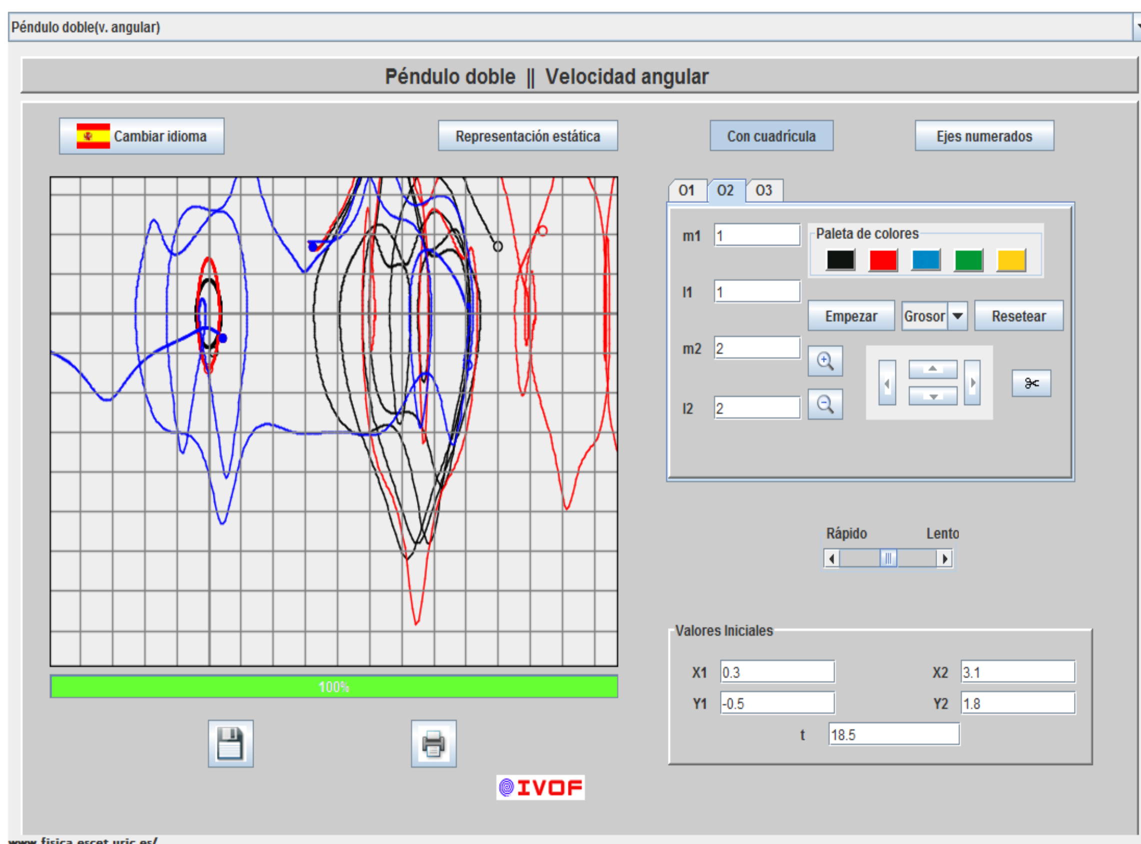
Podemos ver que aunque no se vea el comienzo de la representación que se visualiza con un círculo relleno, se puede ver donde termina cada una de ellas con un círculo normal.



Captura 3.- Representación dinámica del péndulo simple con tres péndulos de distintas masas

En esta última representación se puede ver la representación de tres péndulos caóticos con la X frente a la Y (*Captura 3*). Se puede distinguir perfectamente cada péndulo del otro, gracias no sólo a su distancia entre ellos y su distinto color de masa, sino al tamaño de esas masas y a la longitud del "hilo" que los une a ellas. Además hay que decir, que la representación se hizo en modo dinámico, ya que la barra de progreso muestra un porcentaje del 41%.

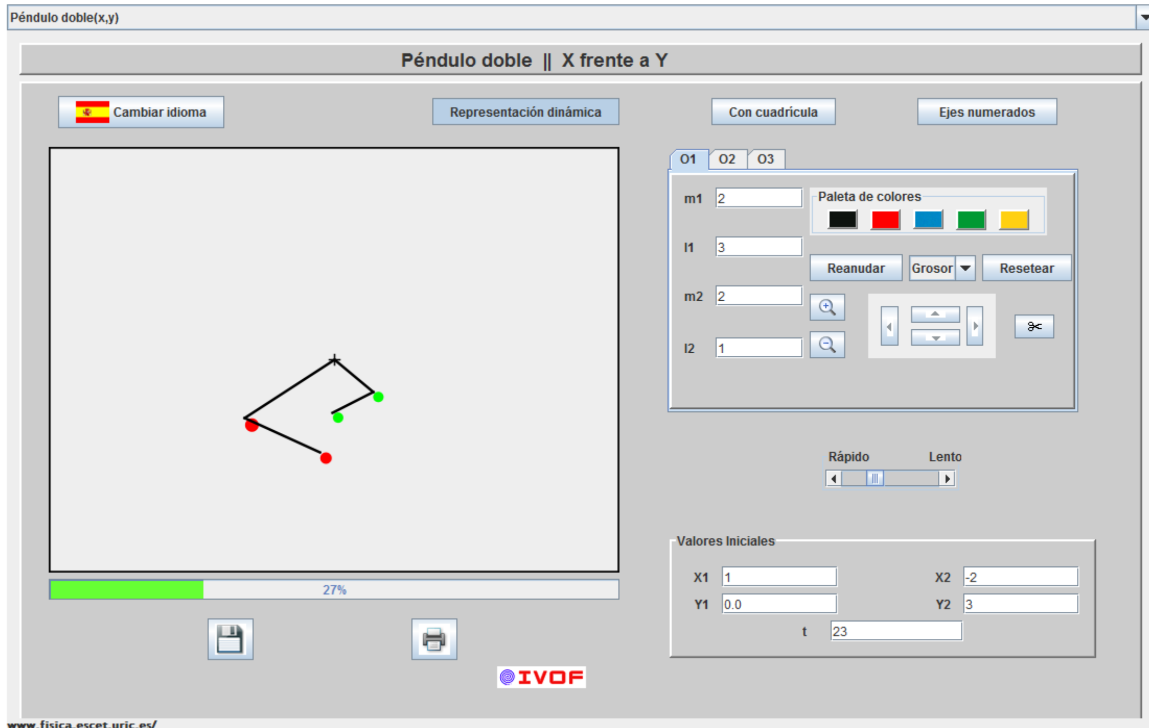
Seguidamente, vamos a observar cómo se comporta el péndulo doble, pero mediante su velocidad angular (*Captura 4*). Podremos observar 3 órbitas bien diferenciadas con su comportamiento caótico propio.



Captura 4.- Representación estática del péndulo doble

Se puede observar que para visualizar mejor los comportamientos de las órbitas y su caos, se ha desplazado la imagen hacia abajo y a la derecha con las herramientas de desplazamiento. Así podemos ver para casi todas las órbitas su comienzo y final representado con los círculos propios.

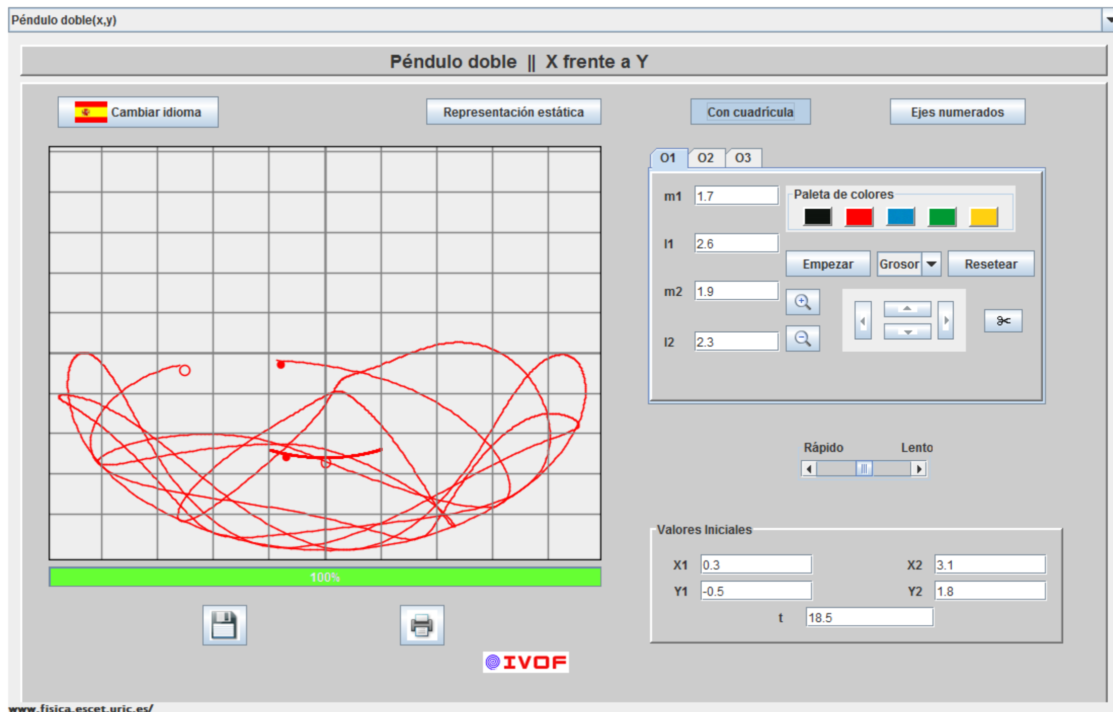
Seguidamente veremos el comportamiento del péndulo doble con la X frente a la Y (*Captura 5*). Podremos ver un par de péndulos dobles que convergen en la misma imagen con distintas masas y longitudes y que aportan una mejor visualización de este fenómeno que la anteriormente descrita (velocidad angular).



Captura 5.- Representación dinámica de dos péndulos dobles

En la imagen perfectamente clara se puede ver la diferencia de masas y longitudes entre los péndulos de cada péndulo doble, y entre ellos.

Al igual que en la anterior captura, podemos ver en la siguiente como se dibujan las trayectorias de un péndulo doble similar.



Captura 6.- Representación estática de un péndulo doble

4. Resultados y conclusiones

Este trabajo ha sido llevado a cabo para determinar los procesos básicos y más complejos para la realización de nuestro *applet*. En cada uno de los apartados se ha intentado dar una imagen sencilla para cualquier lector, y su entendimiento ha sido la meta para la realización del mismo. Cada una de las partes que conforman todo el trabajo ha sido elegida para poder explicar la complejidad de todo el proyecto.

Los resultados conseguidos al final de la realización del *applet* no sólo han sido llegar a mostrar las gráficas de una manera satisfactoria, además:

- Hemos conseguido crear una interfaz gráfica visualmente atractiva y cómoda para el usuario dónde éste podía interpretar cada acción y cada herramienta de la mejor manera posible.

- Hemos descrito de manera visual tres importantes sistemas caóticos: el *Oscilador de Helmholtz*, el *péndulo simple* y el *péndulo doble*. Y no sólo eso, hemos podido dar dos maneras distintas de representación para estos dos últimos, pudiendo elegir entre observar el cambio de la velocidad angular frente al tiempo u observar el péndulo de manera realista al visualizar la X frente a la Y. Además, y para cada uno de los sistemas, hemos podido observar el comportamiento de manera dinámica y como va van oscilando durante el tiempo, incluso variando la velocidad de cada uno de ellos.

- Todo el proyecto ha sido llevado a cabo mediante el lenguaje de programación *Java*. Tanto la elaboración de los métodos que han sabido sacar partido a cada componente como el mecanismo interno con el que se comunicaban las clases entre sí ha sido realizado al milímetro para que el comportamiento de esta herramienta fuera el esperado.

- Finalmente hemos conseguido conocer el funcionamiento del los *threads* para poder simular el comportamiento dinámico de todas los sistemas.

4.1. Aplicaciones futuras

Primeramente, este *applet* como cualquier aplicación desarrollada, debería irse actualizando a los usuarios. Como aplicación que sólo ha sido testada por el propio programador, debería dar paso a que los usuarios pudieran probarla y ver si contiene algún tipo de fallo o se le pudiera añadir algún tipo de herramienta para mejorar lo ya hecho.

Una vez terminado, está aplicación, y como ya dijimos antes, se puede incluir en cualquier página de estudio físico mediante el uso de un archivo en código HTML que lleve. Por ejemplo, en la página web del Departamento de Física de la URJC que

contenga un apartado para todos los *applets* hechos y que supondrían una gran fuente de información.

Como proyecto futuro, se podría introducir con mucha facilidad en este código nuevos sistemas lineales que sirvan para dar más variedad de ecuaciones caóticas al usuario. Además se podría incluir una herramienta que pudiera almacenar todos los resultados obtenidos mediante el método *Rungekutta* en una base de datos para poder estudiarlos de manera matemática.

5. Experiencia personal

Este proyecto para mí ha sido una experiencia importantísima y gratificante en todos los aspectos. He conseguido varias cosas con la realización de este proyecto.

La primera de ellas es haber llevado a cabo este trabajo y poder haber tenido la satisfacción de llevarlo a cabo. Realizar un proyecto de tamaña envergadura sólo por mí mismo sin tener que depender de ayuda externa y dominando aspectos de la programación aprendidos de manera autodidacta me ha enseñado que con trabajo todo puede hacerse.

El siguiente, y bastante importante, es haber realizado este proyecto con el lenguaje de programación *Java* ya que no lo hemos usado en la carrera. Ha sido una gran satisfacción para mí definir un trabajo de esta envergadura con un código del que ido aprendiendo cada día por mi cuenta y por mi propio trabajo. Esta experiencia adquirida me ha demostrado que cualquier lenguaje de programación puede dominarse con trabajo, estudio y sobre todo el empeño necesario. He podido profundizar en el lenguaje Java y entender muchos aspectos de él, empresa muy importante ya que formará parte esencial en mi futura experiencia laboral.

Por último y no menos importante, es haber sabido cumplir todos los requisitos que imponían este trabajo e ir pasando por cada uno de los objetivos que se me ponían delante.

Aunque todo lo dicho es muy bueno, también hay que decir que ha habido momentos malos que como informático tendré que vivir en más ocasiones y han lastrado la velocidad de trabajo. Aún así, gracias al apoyo de mis tutores, amigos y familiares han dado como resultado el térmico de este proyecto.

6. Bibliografía

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

[2] Gregory L. Baker, James A. Blackburn, *The Pendulum: A Case Study in Physics*, Oxford University Press, 2005.

[3] Richard L. Burden, J. Douglas Faires, *Numerical Analysis*, ITP, 1997.

[4] Jesús M. Seoane, Samuel Zambrano, Stefano Euzzor, R. Meucci, F. T. Arecchi, Miguel A.F. Sanjuán: *Avoiding escapes in open dynamical systems using phase control,* Universidad Rey Juan Carlos, 2008.

[5] Troy Shinbrot, Celso Grebogi, Jack Wisdom, James A. Yorke: *Chaos in a double pendulum*. Am. J. Phys.,Vol.60, No.6, 491-499, 1991.

[6] Gabriela González, *Single and Double plane pendulum.*

[7] E.Ott, C.Grebogi and J.A.Yorke; “Controlling Chaos”, Phys. Rev. Lett. 64, 1196 (1190).