Departamento de física

ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE GESTIÓN

Curso Académico 2012/2013

Proyecto de Fin de Carrera

OSCILADOR LIBRE, AMORTIGUADO Y FORZADO EN APPLETS DE JAVA

Autor: Sergio Zapatero Martín

Tutores: Jesús Seoane Sepúlveda

Inés Pérez Mariño

AGRADECIMIENTOS

En este espacio me gustaría agradecer a todas las personas que me han llenado de fuerza y me han hecho ver que el hecho de sacarse una carrera era posible.

Agradecer a mis tutores del proyecto Jesús Seoane Sepúlveda e Inés Pérez Mariño del Departamento de Física de la Universidad Rey Juan Carlos, por la dedicación, el tiempo que me han prestado y la facilidad con que me han hecho aprender las cosas.

Gracias a todos mis compañeros que han hecho posible que recuerde esta etapa con mucha felicidad. Gracias a mi padre, madre, hermano y hermana por estar ahí en los malos y buenos momentos. Nombrar a mis amigos Gabriel Grigelmo Mecerreyes y Rubén Bote García por su ayuda en momentos complicados del proyecto.

Agradecer a todos los miembros del Grupo de Modelado y Realidad Virtual (GMRV) de la Universidad Rey Juan Carlos por su ayuda en el aspecto informático del proyecto.

RESUMEN

Este documento corresponde a la memoria del Proyecto de Fin de Carrera consistente en una aplicación informática que permite la representación del funcionamiento de un sistema físico, concretamente en un oscilador armónico.

El proyecto ha sido realizado en la Universidad Rey Juan Carlos de Móstoles para la carrera de Ingeniería Informática de Gestión y en el departamento de Física. Está desarrollado mediante un *Applet de Java* que permite desarrollar la simulación y comportamiento del oscilador armónico mediante unas variables y parámetros de entrada.

Para su desarrollo se ha utilizado *NetBeans* (*Sun MicroSystems)*. Se trata de un entorno de desarrollo integrado libre, hecho principalmente para el lenguaje de programación *Java*. *NetBeans* se caracteriza por ser un proyecto de código abierto, libre, gratuito y sin restricciones de uso. Además de esto, cuenta con un número importante de módulos para extenderlo.

El hecho de que hayamos decidido optar por un *Applet* para realizar la aplicación tiene que ver con las ventajas que aparecen a continuación. Es multiplataforma, es decir, funcionan en cualquier sistema operativo *(Windows, Linux…),* siempre que exista una máquina virtual de Java. Por otra parte, por la comodidad que ofrece al ser compatible con la mayoría de los navegadores web *(Mozilla Firefox, Internet Explorer, Google Chrome…).* Además, permite al usuario tener acceso completo a la maquina que este usando siempre que el usuario se lo permita.

En cuanto al desarrollo del *Applet*, haremos un desarrollo mas detallado a lo largo de la memoria.

INDICE

1. Introducción: descripción del problema

Un sistema dinámico es un tipo de sistema que presenta un cambio o una evolución de su estado con el paso del tiempo. Los sistemas dinámicos, a su vez, pueden clasificarse en dos tipos, lineales y no lineales.

Un sistema dinámico lineal es aquel en el que el efecto producido es proporcional a la causa.

Un sistema dinámico no lineal es aquel que cambia con el tiempo y puede ser explicado por medio de ecuaciones dinámicas y estructuras matemáticas; o bien, puede ser representado por trayectorias en el espacio de fases caracterizado por su capacidad de percibir la evolución del sistema en el tiempo.

En este proyecto llevaremos a cabo el estudio de un sistema lineal, el oscilador armónico, que hemos presentado en resumen inicial. Para comenzar, indicar que el movimiento oscilatorio es el formado por un objeto de masa m conectado a un muelle horizontal (*Figura 1*) [1].

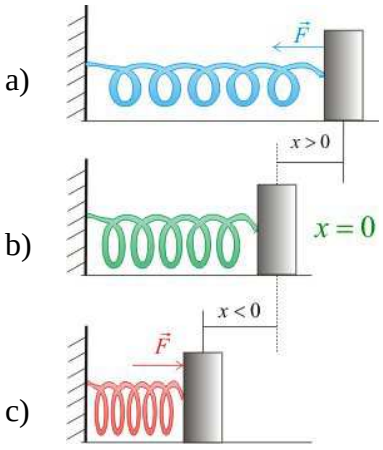

Figura 1. Un bloque unido a un muelle situado sobre una superficie sin

Si el muelle no está estirado o contraído, el objeto está en reposo sobre una superficie sin rozamiento en su posición de equilibrio, la cual queda definida por $x = 0$. Si se mueve el objeto hacia un lado y luego se suelta, oscilará hacia delante y hacía atrás.

Cuando una partícula unida a un muelle ideal sin masa es desplazada hasta una posición x, el muelle ejerce una fuerza cuya magnitud viene dada por la **Ley de Hooke**,

$$F_S = -kx$$

donde k es la constante de recuperación del muelle y F_S es la fuerza ejercida por el muelle. Decimos que la fuerza de la Ecuación es una fuerza de recuperación lineal, ya que es proporcional al desplazamiento con relación a la posición de equilibrio y va dirigida siempre hacia la posición de equilibrio, en sentido opuesto al desplazamiento. Es decir, cuando la partícula es desplazada hacia la derecha (*Figura 1a*), x es positiva y la fuerza ejercida por el muelle tiene una componente negativa, hacia la izquierda. Cuando la partícula se desplaza hacia la izquierda de $x = 0$ (*Figura 1c*), x es negativo y la fuerza ejercida por el muelle tiene una componente positiva, hacia la derecha. Cuando una partícula está bajo el efecto de una fuerza de recuperación lineal, el movimiento de la partícula se corresponde con un tipo especial de movimiento oscilatorio denominado **movimiento armónico simple**. Al sistema que experimenta un movimiento armónico simple se le denomina **oscilador armónico simple**.

En una partícula sometida a una fuerza de recuperación lineal, similar a la de la *Figura 1*, aplicando la segunda ley de Newton en la dirección x tenemos:

$$\sum F = F_S = ma \rightarrow -kx = ma$$

$$a = -\frac{k}{m}x$$

Es decir, **la aceleración es proporcional al desplazamiento de la partícula con relación a la posición de equilibrio y va dirigida en sentido opuesto**. Si desplazamos la partícula hacia una posición $x = A$, y luego cuando está parada la soltamos, su aceleración inicial es $-kA/m$. Cuando la partícula pasa por la posición de equilibrio $x = 0$, su aceleración es cero. En ese instante, su velocidad alcanza un valor máximo, ya que la aceleración cambia de signo. La partícula continúa moviéndose hacia la izquierda de la posición de equilibrio con una aceleración positiva, finalmente alcanza la posición $x = -A$. En ese instante, su aceleración es $+kA/m$ y su velocidad es de nuevo cero. La partícula realiza un ciclo completo de movimiento volviendo a la

posición original, pasando de nuevo por $x = 0$ a la velocidad máxima. Por tanto se observa que la partícula oscila entre los puntos límite $x = \pm A$.

De la siguiente función coseno hay que destacar los siguientes aspectos:

$$x(t) = Acos(\omega t + \emptyset)$$

Los parámetros A, ω y $\emptyset$ son constantes del movimiento. En primer lugar, vemos que el parámetro A, denominado la **amplitud** del movimiento, **es el valor máximo de la posición de la partícula, tanto en la dirección positiva como negativa de la x.** A la contante ω se la denomina frecuencia angular y se mide en rad/s. La frecuencia angular se la puede representar de la siguiente manera:

$$\omega = \sqrt{\frac{k}{m}}$$

Al ángulo constante $\emptyset$ se le denomina **constante de fase** (o ángulo de fase) y, al igual que la amplitud A, queda determinado solamente por la posición y velocidad de la partícula en el instante $t = 0$. Si la partícula está en su posición máxima $x = A$ en el instante $t = 0$, la constante de fase es $\emptyset = 0$ y la representación gráfica del movimiento es la que se muestra en la figura. A la magnitud $(\omega t + \emptyset)$ se la denomina **fase** del movimiento.

1.1 OSCILACIONES AMORTIGUADAS

En los sistemas reales, hay que considerar las fuerzas resistivas, como el rozamiento, que frenan el movimiento del sistema. Como consecuencia, la energía mecánica del sistema disminuye con el tiempo y se dice que el movimiento se amortigua.

Un tipo de fuerza resistiva es proporcional a la velocidad y actúa en dirección opuesta la misma. Dado que la fuerza resistiva puede expresarse como $R = -\mu v$, donde μ es una constante relacionada con la intensidad de la fuerza resistiva, y dado que la fuerza de recuperación que se ejerce sobre el sistema es $-kx$, la segunda ley de Newton nos da

$$\sum F_x = -kx - \mu v = ma_x$$

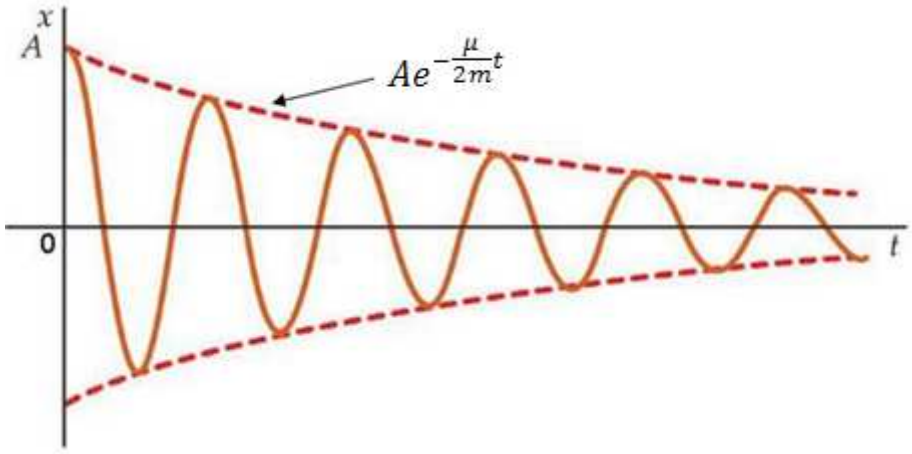

Figura 2. Gráfica de la posición en función del tiempo para un oscilador amortiguado.

Observamos que, **cuando la fuerza resistiva es relativamente pequeña, el carácter oscilatorio del movimiento se conserva, pero la amplitud de la vibración disminuye con el tiempo** y el movimiento, en último término, cesa *(Figura 2)*. A este tipo de sistema se le denomina **oscilador subamortiguado**.

La frecuencia angular de vibración de un sistema amortiguado se expresa de la forma:

$$\omega = \sqrt{\omega_0^2 - \left(\frac{\mu}{2m}\right)^2}$$

donde $\omega_0 = \sqrt{k/m}$ representa la frecuencia angular de oscilación en ausencia de fuerzas resistivas. Es decir, cuando $\mu = 0$, la fuerza resistiva es cero y el sistema oscila con una frecuencia angular ω_0, denominada **frecuencia natural**. A medida que la intensidad de la fuerza resistiva aumenta, las oscilaciones se amortiguan con mayor rapidez. Cuando μ alcanza un valor crítico μ_c, tal que $\mu_c/2m = \omega_0$, el sistema no oscila y se dice que está **críticamente amortiguado**. En este caso, vuelve a la posición de equilibrio siguiendo una curva exponencial en el tiempo *(Figura 3)*.

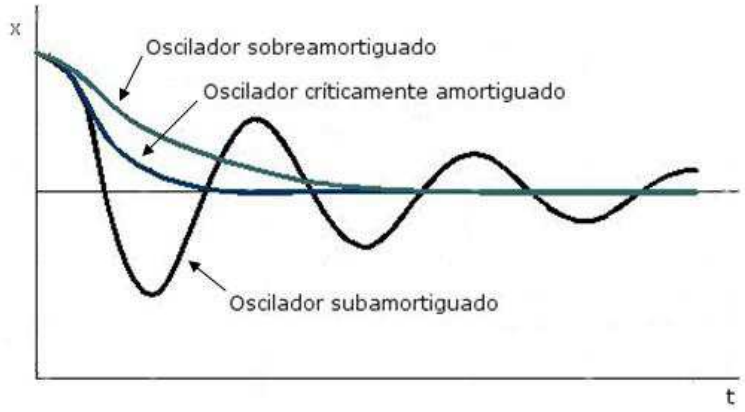

Figura 3. Gráficas de la posición en función del tiempo.

Si el medio fuera altamente viscoso, y los parámetros cumplieran la condición $\mu\ /\ (2m > \omega_0\ )$, el sistema estaría **sobreamortiguado**. Al desplazar el sistema éste no oscila, sino que retorna a la posición de equilibrio. A medida que aumenta la amortiguación, aumenta también el tiempo que necesita la partícula para volver a la posición de equilibrio *(Figura 3)*. Cuando está presente una fuerza resistiva, la energía mecánica del oscilador acaba disminuyendo hasta llegar a cero. Esta energía mecánica se transforma en energía interna del sistema oscilador y del medio resistivo.

1.2 OSCILACIONES FORZADAS

Se puede transferir energía al sistema aplicando una fuerza que actúe en el sentido del movimiento del oscilador. Por ejemplo, se puede mantener un niño balanceándose en un columpio dándole oportunos "empujones" en los momentos adecuados. La amplitud del movimiento permanece constante si la energía que se aporta en cada ciclo de movimiento es exactamente igual a la pérdida de energía mecánica en cada en cada ciclo debido a las fuerzas resistivas.

Un ejemplo habitual de oscilador forzado es un oscilador amortiguado al que se le comunica una fuerza externa que varía periódicamente como, por ejemplo, $F(t) = F_0 sen\ \omega t$, donde ω es la frecuencia angular de la fuerza externa y F_0 es una constante. La frecuencia ω de la fuerza externa es diferente de la frecuencia natural del oscilador ω_0.

La solución al sistema amortiguado es la siguiente:

$$x = Acos(\omega t + \emptyset)$$

donde

$$A = \frac{F_0/m}{\sqrt{(\omega^2 - \omega_0^2)^2 + \left(\frac{\mu\omega}{m}\right)^2}}$$

y donde $\omega_0 = \sqrt{k/m}$ representa la frecuencia natural del oscilador no amortiguado $(\mu = 0)$. La ecuación muestra que la amplitud del oscilador forzado es constante para una fuerza externa dada, porque es esa fuerza externa la que conduce al sistema a un estado estacionario. En el caso de una amortiguación de pequeña intensidad, la amplitud se hace muy grande cuando la frecuencia de la fuerza externa se aproxima a la frecuencia natural de oscilación, esto es, cuando $\omega \approx \omega_0$. Al drástico incremento en la amplitud cerca de la frecuencia natural se le denomina **resonancia**, y a la frecuencia natural ω_0 se la denomina también **frecuencia de resonancia** del sistema *(Figura 4).*

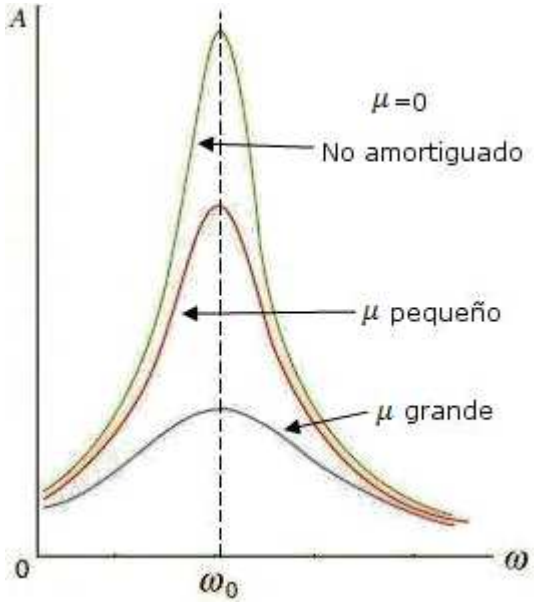

Figura 4. Curva de resonancia

Se puede observar que la amplitud se incrementa al disminuir la amortiguación $(b \to 0)$ y que la curva de resonancia se aplana a medida que la amortiguación aumenta. En ausencia de una fuerza de amortiguación $(\mu = 0)$, de la ecuación *(Figura 4)* deducimos que la amplitud en el estado estacionario se aproxima a infinito cuando

$\omega \to \omega_0$. Es decir, si no hay fuerzas resistivas en el sistema y seguimos forzando el oscilador mediante una fuerza sinusoidal a la frecuencia de resonancia, la amplitud del movimiento crecerá de forma ilimitada.

2. Objetivos

A continuación, se enumerarán los objetivos a realizar:

- Para poder realizar una buena simulación del tipo de sistema físico, hay que obtener los conocimientos teóricos necesarios. Tenemos que conocer las ecuaciones y los tipos de gráfica con las que vamos trabajar y saber interpretar el resultado obtenido a partir de unos/as variables/parámetros de entrada.

- En segundo lugar, será necesario el conocimiento del lenguaje *Java* y la familiarización con el entorno de programación que vamos utilizar, en nuestro caso *NetBeans*. Tendremos que adquirir los conceptos básicos de la programación orientada a objetos (POO) [3] y de los *Applets de Java* [5].

- La implementación debe funcionar correctamente y permitirá obtener distintos tipos de representación. Las principales opciones de representación serán las siguientes:

    - Representación de una o dos órbitas.

    - Representación estática o dinámica.

    - Modificación del color tanto para la gráfica 1, como para la gráfica 2.

    - Representación gráfica mediante los ejes $X\dot{X}$, XT o representación del muelle y eje XT.

    - Representación de la curva de resonancia.

  Además, el usuario podrá iniciar la aplicación, detenerla, pausarla en el momento que crea oportuno. También contará con las opciones de aumentar el grosor, guardar la representación, imprimir, cortar, zoom y desplazamiento.

- Para terminar, habrá que realizar una memoria que contenga toda la información acerca del *Applet* desarrollado. Este documento contendrá los objetivos a realizar, una descripción de los componentes utilizados en el *Applet*, la manera en que esta implementado las funciones en el programa, desarrollo teórico del sistema simulado...

2.1 Estudio de las alternativas

En este proyecto, el sistema presenta solución exacta, pero para dar más robustez a la aplicación, tenemos decidido usar la técnica numérica para que el usuario docto pueda modificar el código fuente e introducir sistemas nuevos.

2.1.1 El método matemático

En general, los sistemas dinámicos lineales no pueden resolverse sin utilizar métodos numéricos [4] que integren las ecuaciones que describen dicho sistema y que nos ayudan a aproximarnos a una solución real. Estos métodos son los que se utilizan para la resolución de ecuaciones del siguiente tipo:

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

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

2.1.2 El método de Euler

En nuestro caso, tratamos de utilizar dos métodos. Primero recurrimos al Método de *Euler*, llamado así en honor a **Leonhard Paul Euler** *(Figura 5)*, que es un procedimiento de integración numérica para resolver ecuaciones diferenciales ordinarias a partir de un valor inicial dado.

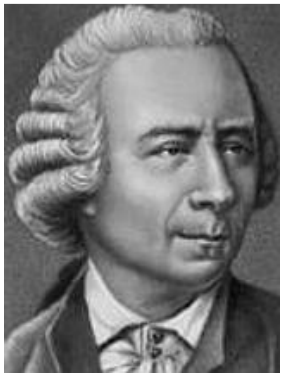

Figura 5. Leonhard Paul Euler.

No es un método demasiado complejo, pero en el caso de nuestro problema, encontrar una solución utilizando esta metodología se nos plantea complicado por su escasa exactitud *(Figura 6)* [2].

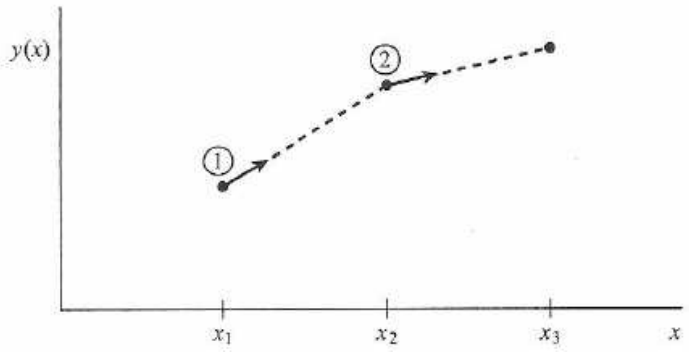

Figura 6. Representación gráfica del método de Euler.

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

$$Nuevo\ valor\ =\ Valor\ anterior\ +\ Pendiente\ \times\ Tamaño\ de\ Paso$$

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

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

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

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

2.1.3 El método del punto medio

Este método no es sino una mejora del método de *Euler*. Cuenta con una exactitud de segundo orden, mayor que la del método explicado anteriormente y obtenido mediante la derivada inicial de cada paso con el fin de encontrar el punto medio en cada uno de los intervalos [2]. En la siguiente gráfica *(Figura 7)*, los puntos rellenos indican los valores reales de la función, y los huecos aquellos que se descartan tras el cálculo y uso de sus derivadas.

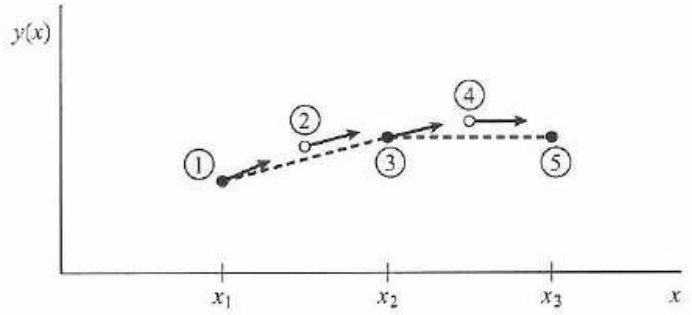

Figura 7. Representación del método del punto medio.

2.1.4 El método de Runge-Kutta

Es el método que utilizamos para dar solución a nuestro problema. Al igual que los anteriores, es un método genérico enfocado a la resolución de ecuaciones diferenciales y que cuenta con una exactitud muy alta [2]. Esta metodología data de principios del siglo XX y fue desarrollada por los matemáticos Carl Runge y Martin Wilhelm Kutta *(Figura 8)*.

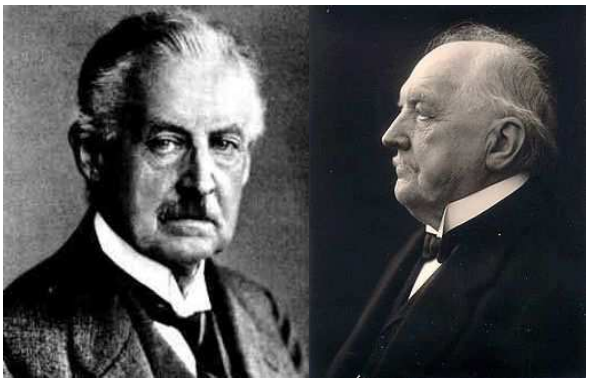

Figura 8. Carl Runge y Martin Wilhelm Kutta.

Realizar los cálculos manualmente es muy complicado, pero cuenta con la ventaja de que su programación para ordenadores es sencilla. Podríamos decir que a su vez se divide en cuatro submétodos. Se elige una anchura de paso *h*, y se calculan cuatro valores. El primero de ellos encontraría un punto inicial (1), posteriormente se llegaría a dos puntos medios (2,3) y luego al que podría ser un posible punto final (4). Después de esto, y a partir de estas derivadas se calcula el valor de la función a la que querríamos llegar (yn+1) *(Figura 9).*

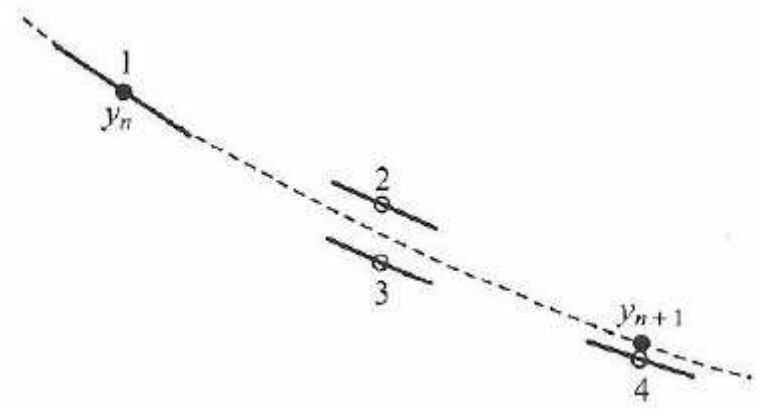

Figura 9. Representación gráfica del método Runge-Kutta de cuarto orden.

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

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

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

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

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

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

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

2.2 Metodología de desarrollo

Antes de indicar la metodología empleada definiremos el significado de Ingeniería del Software. Según *Bohem*, la ingeniería del Software es la aplicación práctica del conocimiento científico en el diseño y construcción de programas de computadora y la documentación asociada requerida para desarrollar y operar (funcionar) y mantenerlos. Así como también desarrollo de software o producción de software.

En el caso del desarrollo de este *Applet* se ha optado por la metodología incremental *(Figura 10)*. Las ventajas que presenta esta metodología es que al ser un paradigma incremental se reduce el tiempo de desarrollo inicial. También provee un impacto ventajoso frente al cliente, que es la entrega temprana de partes operativas del Software. Resulta más sencillo acomodar cambios al acotar el tamaño de los incrementos. Por su versatilidad requiere de una planeación cuidadosa tanto a nivel administrativo como técnico.

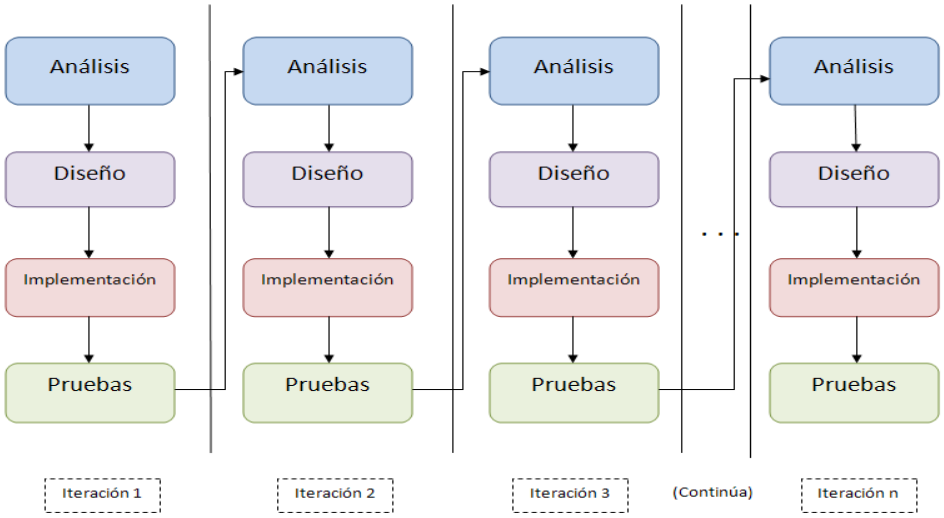

Figura 10. Modelo incremental.

Fases del modelo incremental:

- Análisis: en esta etapa se debe entender y comprender de forma detallada cual es la problemática a resolver, verificando el entorno en el cual se encuentra dicho problema, de tal manera que se obtenga la información necesaria y suficiente para afrontar su respectiva solución.

- Diseño: una vez que se tiene la suficiente información del problema a solucionar, es importante determinar la estrategia que se va a utilizar para resolver el problema.

- Implementación: partiendo del análisis y diseño de la solución, en esta etapa se procede a desarrollar el correspondiente programa que solucione el problema mediante el uso de una herramienta computacional determinada.

- Pruebas: los errores humanos dentro de la programación de los computadores son muchos y aumentan considerablemente con la complejidad del problema. Cuando se termina de escribir un programa de computador, es necesario realizar las debidas pruebas que garanticen el correcto funcionamiento de dicho programa bajo el mayor número de situaciones posibles a las que se pueda enfrentar.

3. Descripción informática

Para la realización de la aplicación lo más conveniente ha sido implementarlo mediante un *Applet de Java*. A continuación, llevaremos a cabo una breve explicación de lo que es un *Applet*.

Un *Applet de Java* es un código *Java* que carece de un método *main*, por eso se utiliza principalmente para el trabajo de páginas web, ya que es un pequeño programa que es utilizado en una página HTML y representado por una pequeña pantalla gráfica dentro de ésta.

Las ventajas que ofrecen son, entre otras, que es multiplataforma, es decir, funcionan en cualquier sistema operativo (Windows, Linux…), siempre que exista una máquina virtual de Java. Por otra parte, ofrece una compatibilidad con la mayoría de los navegadores web *(Mozilla Firefox, Internet Explorer, Google Chrome…).* Además, permite al usuario tener acceso completo a la maquina que este usando siempre que el usuario se lo permita.

3.1 Especificación

La especificación de los requisitos es una descripción del comportamiento del *Applet* que se ha desarrollado. Contiene dos tipos de requisitos, funcionales y no funcionales, que se describirán a continuación.

3.1.1 Requisitos funcionales

Son las características requeridas del sistema o aplicación, como acciones que deberá poder realizar. Mencionar las siguientes características:

- El usuario podrá establecer las condiciones iniciales y parámetros para el tipo de oscilador armónico que elija.
- El programa deberá representar la evolución temporal de la ecuación asociada al oscilador armónico y sus tipos, tanto para una gráfica como para dos.
- El programa permitirá representar el movimiento del muelle.

- El programa permitirá representar la curva de resonancia.
- Se podrá elegir el tipo de representación temporal, eligiendo entre estática y dinámica, en la que existirá una barra que permitirá establecer la velocidad con la que es dibujada (sólo tendrá funcionalidad para la representación dinámica).
- Tanto para la representación estática como para la dinámica existirán herramientas de zoom y de desplazamiento para mejorar la visión de la gráfica.
- El usuario podrá elegir el color que desee para representar la gráfica.
- El usuario podrá establecer el grosor que desee para representar la gráfica.
- El usuario podrá iniciar, parar y pausar la representación de las gráficas.
- El programa deberá poder imprimir el panel de representación del movimiento.
- El programa deberá poder guardar una imagen de la representación del movimiento.
- El programa deberá poder cortar los puntos no deseados por el usuario.

3.1.2 Requisitos no funcionales

Son requisitos que imponen restricciones en el diseño o la implementación. Son propiedades o cualidades que el producto debe tener. Cabe destacar las siguientes:

- La aplicación deberá estar desarrollada mediante un *Applet en Java*.
- El *Applet* deberá estar integrado en un navegador web.
- El *Applet* deberá representar el tipo de gráfica de forma rápida y precisa.
- El *Applet* debe tener un tamaño adecuado para que se visualice por completo en la pantalla.
- La aplicación debe ser sencilla y manejable para el usuario, sin que suponga problemas al utilizarlo.

- El programa debe ser robusto de manera que controle las excepciones e informe al usuario del error producido.

3.2 Diseño

3.2.1 Entorno de desarrollo, lenguaje de programación y bibliotecas

El entorno de desarrollo utilizado es *NetBeans*. La razón de su uso es que a la hora de diseñar interfaces gráficas proporciona una gran manejabilidad. En cuanto al lenguaje de programación utilizado es *Java*, como hemos comentado anteriormente.

Las bibliotecas más utilizadas son las siguientes: *AWT*, *Swing*, *IO*, *Util*, *Print*.

AWT se ha utilizado para establecer el color de las gráficas, obtener el entorno gráfico, establecer el tipo de fuente, crear imágenes, entre otras mas funciones.

Swing es una biblioteca gráfica para *Java*. Mediante esta biblioteca hemos utilizado cajas de texto, botones, desplegables, etc., para dar la apariencia a nuestra interfaz. Swing ha sido totalmente escrito en Java utilizando el paquete *AWT*, y pone a disposición del usuario muchas clases que están también en *AWT*, pero mucho mejor y más potente. Además introduce muchas más clases que no están en *AWT*.

La biblioteca *IO* nos servirá para la entrada y salida a través de flujos de datos y ficheros del sistema.

La biblioteca *Util* es necesaria a la hora de utilizar los *ArrayList* que contienen los datos necesarios para realizar la gráfica. Y, por último, la biblioteca *Print* nos proporcionará los métodos necesarios para poder imprimir el panel de representación.

A continuación se llevará a cabo una descripción de todos los componentes utilizados en la aplicación:

- **jPanel**: es un objeto de los llamados "contenedores". Sirven para contener otros objetos. Actúan como "cajas" en las que es posible insertar elementos y así poderlos manejar como una agrupación.

- **jLabel**: etiqueta que nos permite mostrar un texto.
- **jTextBox**: permite al operador del programa ingresar una cadena de caracteres por teclado.
- **jButton**: se trata de un botón que permite al usuario comenzar un evento, como buscar, aceptar una tarea, interactuar con un cuadro de diálogo…
- **jCheckBox**: es un elemento de interacción de la interfaz gráfica que permite a éste hacer selecciones múltiples de un conjunto de opciones.
- **jProgressBar**: nos mide el progreso de una tarea o proceso en nuestro programa.
- **jSlider**: es un elemento de las interfaces gráficas que permite seleccionar un valor moviendo un indicador o, en algunos casos, el usuario puede hacer clic sobre algún punto del slider para cambiar hacia ese valor.
- **jComboBox**: elemento que se despliega hacia abajo y nos permite elegir entre varias opciones diferentes.
- **jSpinner:** es un elemento de las interfaces gráficas que permite al usuario ajustar un valor dentro de un cuadro de texto adjunto a dos flechas que apuntan en direcciones opuestas (generalmente una hacia arriba y otra hacia abajo).
- **jColorChooser:** es una clase que proporciona a los usuarios una paleta de colores donde pueden elegir un color.
- **JFileChooser**: permite mostrar una ventana para que el usuario navegue por los directorios y elija un fichero. En este proyecto únicamente podrá usarse para salvar el fichero.

3.2.2 Interfaz del applet

A la hora de elaborar la aplicación hay que decidir qué tipo de *Layout* hay que utilizar. El *Layout* es la clase que decide cómo se reparten los botones dentro de la ventana. Esta clase decide en qué posición van los botones y los demás componentes, si

van alineados, en forma de matriz, cuáles se hacen grandes al agrandar la ventana, etc. Otra cosa importante que decide el *Layout* es qué tamaño es el ideal para la ventana en función de los componentes que lleva dentro. En nuestro caso hemos decido elegir *Free Design*. La razón de esta elección es que es adaptativo, flexible y muy sencillo de usar, aunque tienes que asegurarte que tus componentes reaccionen apropiadamente a los cambios de tamaño en las ventanas *(Figura 11)*.

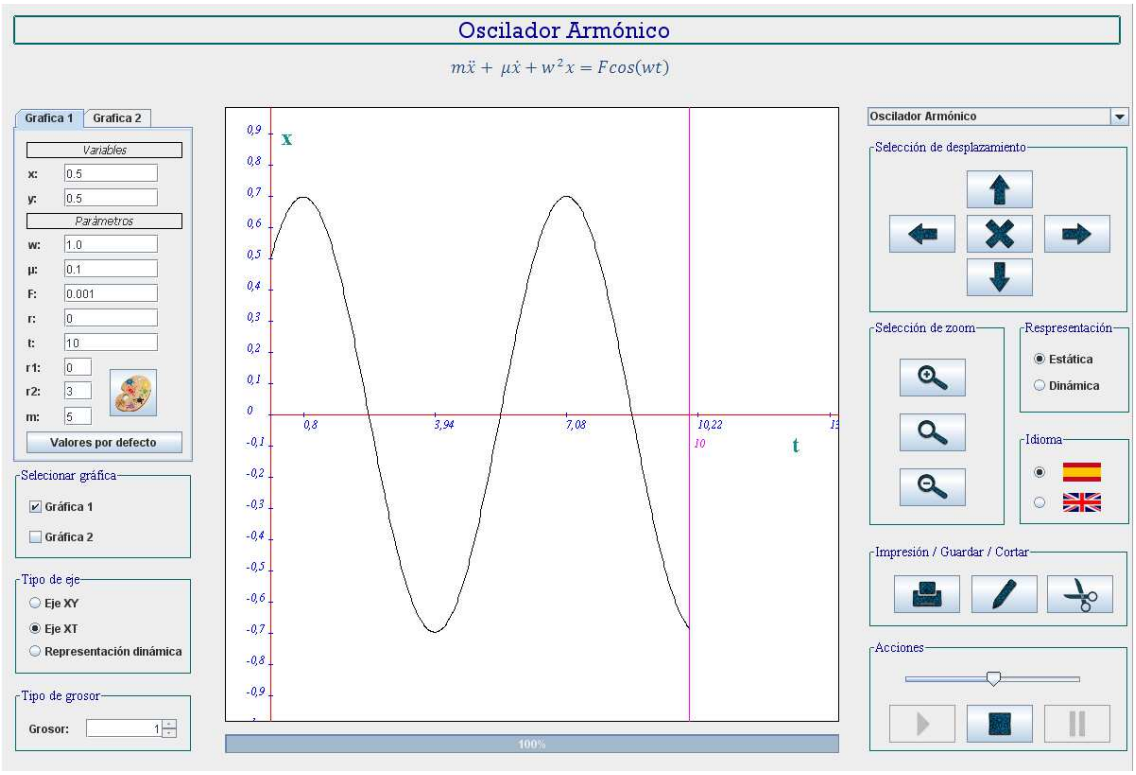

Figura 11. Visión general del applet.

3.2.2.1 Panel encabezado del *Applet*

Se trata de dos *jLabel* que pertenecen al *jPanel* general. Uno del ellos contendrá el título e irá bordeado y el *jLabel* de abajo contendrá la ecuación del movimiento *(Figura 12)*.

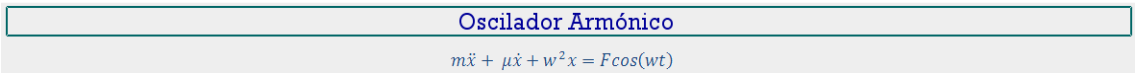

Figura 12. Encabezado del applet.

3.2.2.2 Panel de introducción de parámetros

Está compuesto por un *jTabbedPane* con dos pestañas. Cada pestaña representa los datos de la grafica 1 y de la gráfica 2. Está compuesto por diez *jTextField*, doce *jLabel* y dos *jButton* en cada pestaña. Este panel permite al usuario poder establecer los valores de los parámetros y variables que considere necesarios, elegir el color con el que quiere que se represente la gráfica y restablecer los valores por defecto es de las/los variables/parámetros *(Figura 13)*.

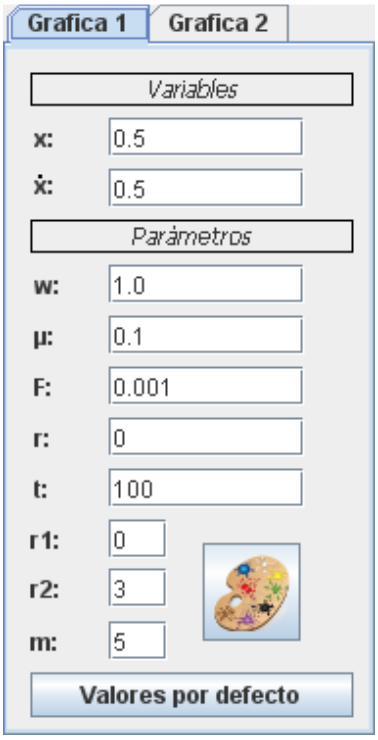

Figura 13. Panel de introducción de parámetros.

3.2.2.3 Panel de selección de gráfica

Se compone de dos *JCheckBox* que permite al usuario seleccionar la gráficas que desee visualizar por pantalla *(Figura 14)*.

Figura 14. Panel de selección de gráfica.

3.2.2.4 Panel de selección del tipo de eje

Compuesto por tres *JRadioButon*. El usuario podrá elegir en qué tipo de eje quiere que se represente la gráfica, seleccionando una de las tres opciones que son: eje $X\dot{X}$, eje XT y representación dinámica (que mostrará el muelle y la gráfica XT) *(Figura 15)*. Nota: este panel será inutilizable si el usuario elige la opción de curva de resonancia.

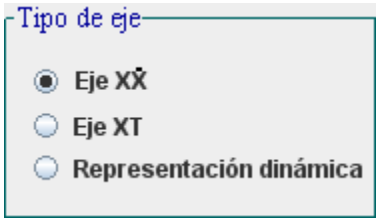

Figura 15. Panel de selección del tipo de eje.

3.2.2.5 Panel de selección del tipo de grosor

Se compone de un *JSpiner* y un *jLabel*. El *jSpiner* permite al usuario poder elegir el grosor de la gráfica *(Figura 16)*. Nota: el grosor máximo será 4.

Figura 16. Panel de selección del tipo de grosor.

3.2.2.6 Selección del tipo de oscilador

Se trata de un *jComboBox* en el que el usuario podrá elegir el tipo de oscilador que desee visualizar. Entre las opciones están: oscilador armónico, oscilador amortiguado, oscilador forzado y curva de resonancia *(Figura 17)*.

Figura 17. Combo Box de selección del tipo de oscilador.

3.2.2.7 Panel de selección de desplazamiento

Este panel está compuesto por 5 *JButton (Figura 18)*, entre los cuales se encuentran:

Figura 18. Panel de selección de desplazamiento.

- Botón  : permite desplazar la gráfica hacia arriba.
- Botón  : permite desplazar la gráfica hacia la izquierda.
- Botón  : permite desplazar la gráfica hacia la derecha.
- Botón  : permite desplazar la gráfica hacia abajo.
- Botón  : sitúa la gráfica en su posición de origen (por defecto).

3.2.2.8 Panel de selección de zoom

Se compone de tres *JButton (Figura 19)*:

Figura 19. Panel de selección de zoom.

- El botón "acercar zoom"  permitirá al usuario aumentar de tamaño la gráfica.
- El botón "alejar zoom"  permitirá al usuario disminuir el tamaño de la gráfica.
- El botón "zoom por defecto"  establecerá el zoom original de la gráfica.

3.2.2.9 Panel de selección del tipo de representación

Está compuesto por dos *JRadioButton (Figura 20)*. El usuario podrá seleccionar si la representación de la gráfica es estática o dinámica. Existe una excepción, y es que si el usuario en el panel de tipo de eje elige la opción de representación dinámica, no podrá, a su vez, representarla estáticamente.

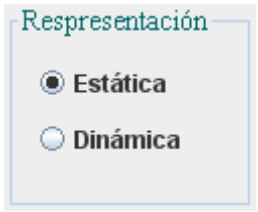

Figura 20. Panel de selección del tipo de representación.

3.2.2.10 Panel de selección del tipo de idioma

Este panel se compone de dos *JRadioButton* y dos *jLabel*. Estos *jRadioButton* permiten al usuario seleccionar uno de los dos idiomas, es decir, inglés o español *(Figura 21)*.

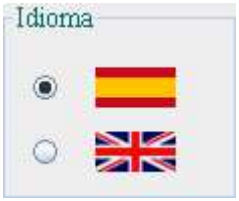

Figura 21. Panel de selección del tipo de idioma.

3.2.2.11 Panel de impresión, guardado y cortado

Está compuesto por tres *jButton (Figura 22)*:

Figura 22. Panel de selección para las opciones imprimir, guardar y cortar.

- El botón de impresión  abrirá una nueva ventana en la que aparecerán los parámetros de impresión que el usuario podrá elegir y permitirá usuario la impresión de la gráfica.

- El botón guardar  generará un archivo de formato *".png"* con la representación del sistema elegido.

- El botón "tijeras"  permitirá al usuario cortar la trayectoria de la gráfica en el momento en que este accione el botón (representación dinámica).

3.2.2.12 Panel para iniciar, parar y pausar gráfica

Este panel está compuesto por un *JSlider* y tres *JButton*. El *JSlider* permitirá al usuario poder establecer la velocidad de la representación *(Figura 23)*. Sólo surtirá efecto si la representación es dinámica.

En cuanto a los tres *JButton*, se encuentran los siguientes:

- Botón "*play*"  : botón encargado de iniciar la representación.
- Botón "*stop*"  : botón por el que se termina la representación.
- Botón "*pause*"  : botón encargado de detener la representación. Este botón solo podrá usarse cuando se trate de una representación dinámica.

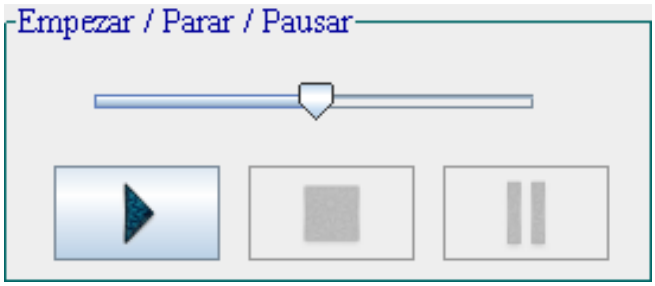

Figura 23. Panel para seleccionar el tipo de acción.

3.2.2.13 Barra de progreso

Se trata de un *jProgressBar* que se encarga de mostrar el progreso de la representación de la gráfica *(Figura 24)*.

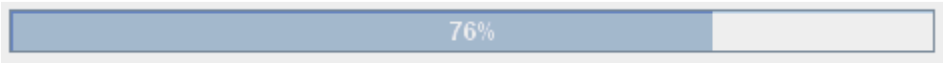

Figura 24. ProgressBar encargada de representar el progreso.

3.2.2.14 Panel de representación de la gráfica

Este panel de dibujo se trata de un *jPanel*. Inicialmente aparecerá en blanco, y no cambiará hasta que el usuario accione el botón "*play*" (siempre que los parámetros/variables sean correctos). En él se podrá ver la representación del movimiento elegido por el usuario *(Figura 25)*.

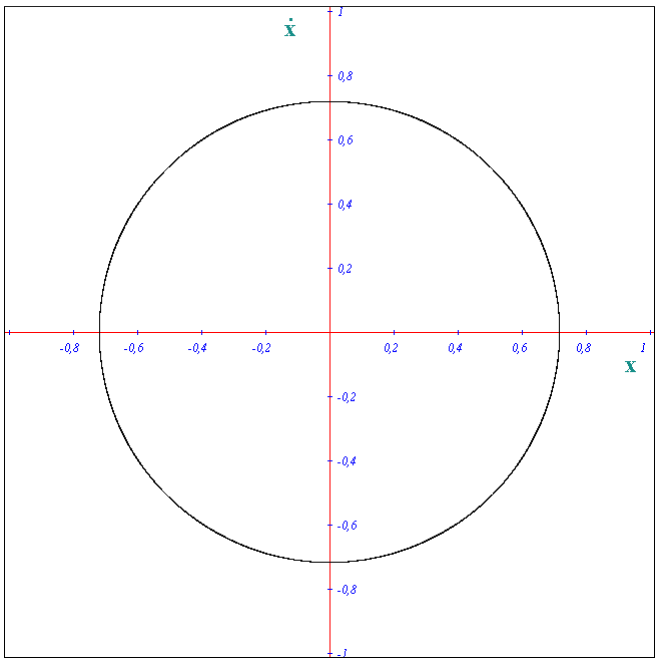

Figura 25. Panel de representación de la gráfica.

3.2.2.15 Panel para confirmar el guardado de la imagen

Se trata de un *JOptionPane*. Este método muestra una ventana pidiendo una confirmación al usuario y da la opción de "aceptar" o "cancelar" el guardado de la imagen de representación de la gráfica *(Figura 26)*.

Figura 26. Panel para confirmar el guardado de la imagen.

3.2.2.16 Diálogo de imagen generada con éxito

JDialog compuesto por un *JPanel* y dos *JLabel*. Esta ventana de diálogo será la encargada de informar al usuario que la creación de la imagen ha sido correcta. *(Figura 27)* [8].

Figura 27. Diálogo que muestra éxito al generar la imagen.

3.2.2.17 Panel de error al introducir los datos

JDialog compuesto por un *JPanel* y dos *JLabel*. Aparecerá cuando el usuario introduzca incorrectamente los parámetros o las variables. En este diálogo se encontrará el nombre de aquellos que estén introducidos incorrectamente y la gráfica a la que corresponde *(Figura 28)* [8].

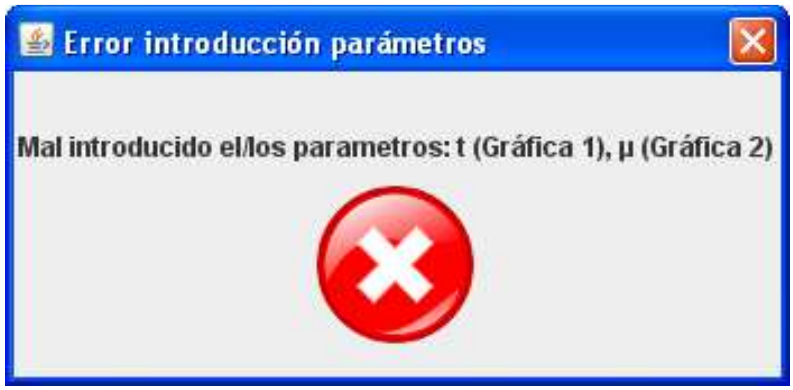

Figura 28. Diálogo que muestra el error al introducir los parámetros.

3.2.2.18 Panel de impresión

Se trata de un *printDialog* perteneciente a la clase *PrinterJob*. Esta ventana permitirá al usuario poder elegir la impresora con la que va a realizar la impresión del panel de representación. Se podrá elegir el número de copias y cada impresora tendrá una serie de propiedades que se podrán establecer en el botón "Properties" situado en la parte superior derecha. Mencionar que si el usuario tiene instalado el programa "*PDF Creator*" también podrá imprimirlo de esta forma *(Figura 29)*.

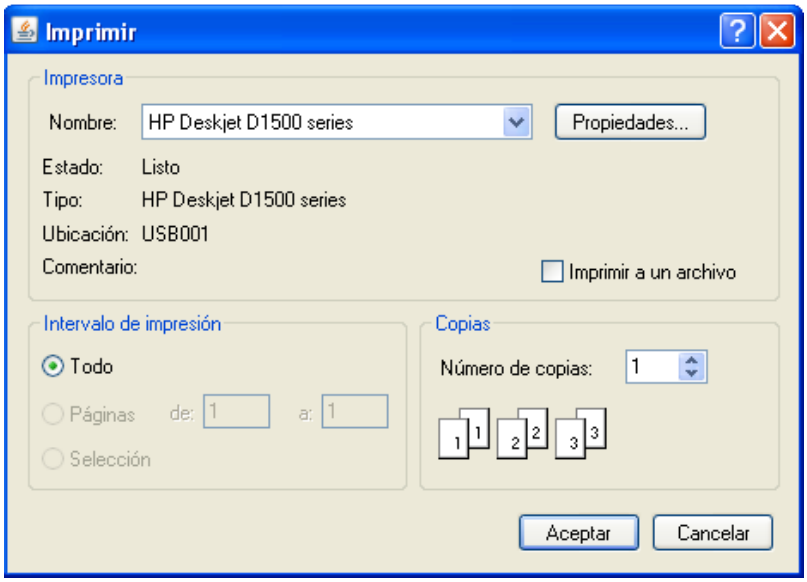

Figura 29. Diálogo de impresión.

3.2.2.19 Panel para el guardado de la imagen (*JFileChooser*)

Panel que aparecerá cuando el usuario decida que quiere salvar la imagen de la representación. Mediante este componente, el usuario podrá establecer la ruta en donde desee guardar el archivo, establecer su nombre y salvarlo con el formato *".png"*. El componente contará con un filtro para visualizar los archivos que sean *".png" (Figura 30)*.

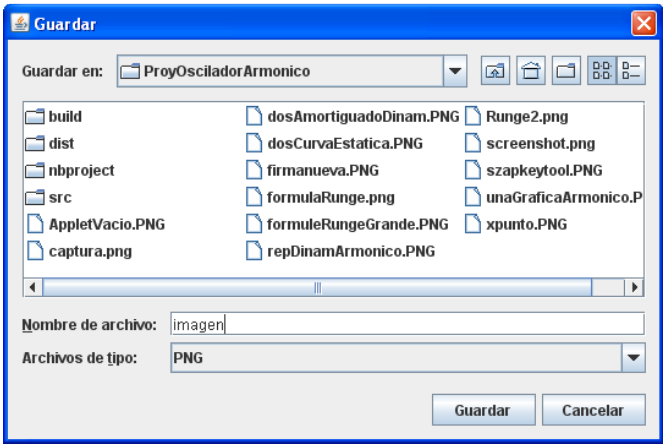

Figura 30. Panel guardar la imagen.

3.2.3 Diseño de clases

A continuación se explicará brevemente la función que realiza cada una de las clases utilizadas y la estructura que se ha seguido para su realización *(Figura 31)*.

- **Runge**: clase encargada que realizar el método de *Runge-Kutta* que es un método genérico de resolución numérica de ecuaciones diferenciales.

- **OsciladorArmonico**: extiende de la clase *Runge* y es la encargada de realizar los cálculos asociados a la representación del oscilador armónico. Como particularidad, contiene la variable ω, que es la frecuencia angular del oscilador armónico.

- **OsciladorAmortiguado**: extiende la de la clase *OsciladorArmonico* y es la encargada de realizar los cálculos asociados a la representación del oscilador amortiguado. Contiene la variable μ que corresponde al coeficiente de rozamiento del oscilador.

- **OsciladorForzado**: extiende de la clase *OsciladorAmortiguado* y es la encargada de realizar los cálculos asociados a la representación del oscilador amortiguado. Como parámetros particulares del oscilador se encuentran F, (fuerza ejercida sobre el oscilador) y r (frecuencia de forzamiento externo).

- **AppletOscilador**: clase que hereda del componente *JApplet*, pudiendo utilizar sus métodos. Se trata de la clase principal de la aplicación y es la encargada de

iniciar la ejecución. Cuenta con el método *init()* que contiene la instrucciones para inicializar el *Applet*. Juega un papel muy importante en esta clase los *Threads* [6]. La creación de un *Thread (hilo)* es una característica que permite a una aplicación realizar varias tareas a la vez (concurrentemente). Los hilos de ejecución comparten una serie de recursos tales como el espacio de memoria, los archivos abiertos, situación de autenticación, etc.

- **PintarGrafica**: hereda de la clase *Runnable* con su correspondiente método *public void run()*. Una de las razones del uso de esta interfaz es que se puede usar fácilmente para crear hilos trabajadores. El objetivo primordial de esta clase es representar el comportamiento estático y dinámico, el movimiento del muelle y la curva de resonancia.

- **Parametros**: clase que contiene los parámetros de todos los tipos de osciladores. En ella podemos observar que existe un constructor en el que se le pasará todos los parámetros del oscilador. También contamos métodos de acceso a los atributos de la clase (*getters* y *setters*).

- **VarRunge**: clase que cuenta con las variables que han sido usadas en la clase *Runge*. Contará con un constructor y métodos *getters* y *setters* de las variables x, y y t.

- **OperacionesAuxiliares**: clase en la que se encuentran los métodos auxiliares que serán esenciales para la realización del proyecto. Cabe destacar los siguientes: *tamanioFuente*, *tamanioFuenteEjes*, *tamanio*, *tamanioResonancia*, *dialogExito*…

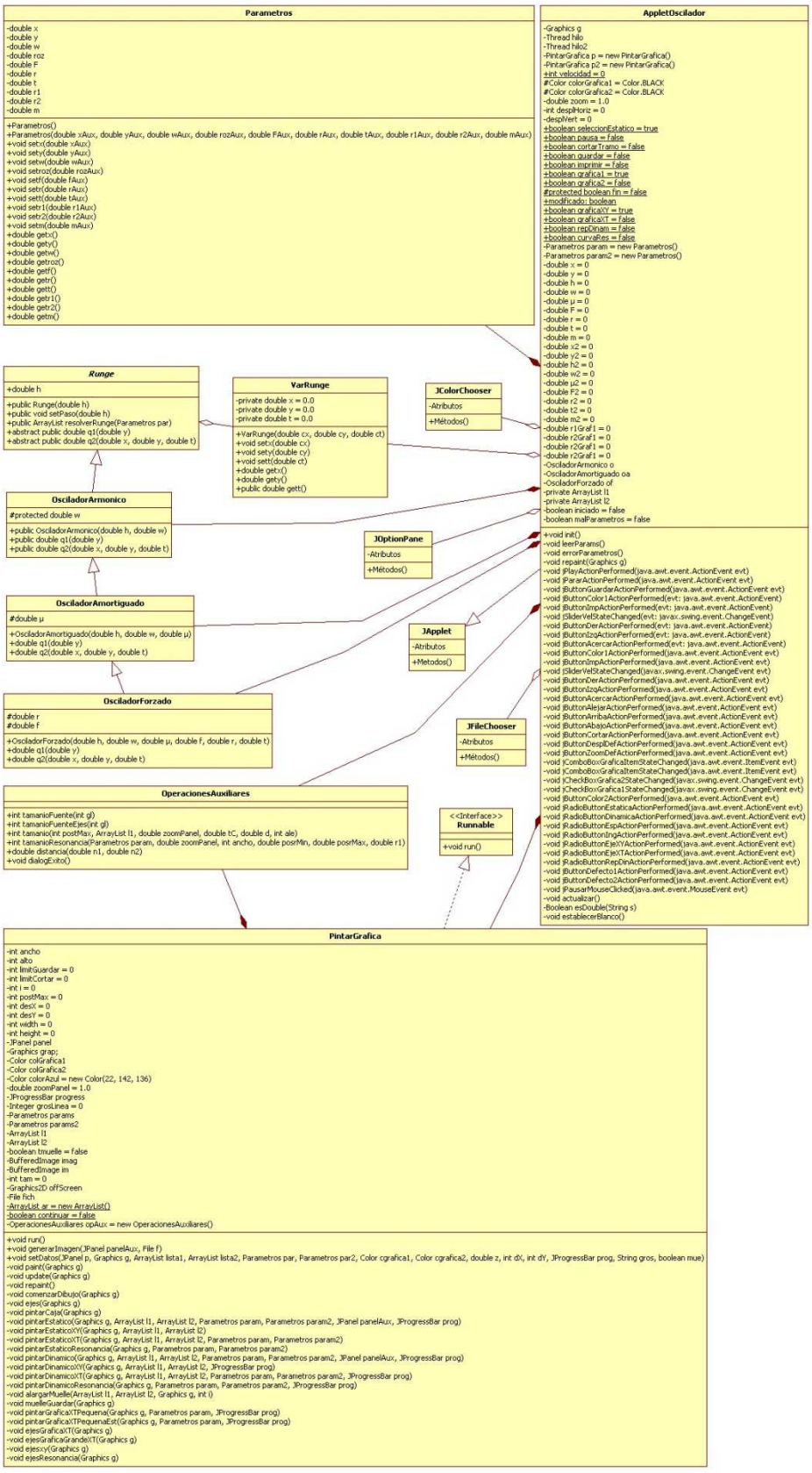

Figura 31. Diseño estático de clases

3.3 Modo de uso

Al comenzar la ejecución de la aplicación, lo primero que debemos hacer es introducir las variables y parámetros de forma correcta en el panel situado en la parte superior izquierda. Estas/os variables/parámetros serán indispensables para la representación de cualquier tipo de oscilador (cada tipo usará unos valores/parámetros u otros). También se podrá establecer el color de la gráfica mediante el botón de la paleta de colores y establecer las condiciones iniciales de las/los variables/parámetros con el botón por defecto. Otra de las opciones que permite la aplicación es la de poder representar dos órbitas a la vez.

Por otra parte, podremos decidir si queremos que la representación de la órbita sea de forma estática o dinámica. Si optamos por representar la gráfica de forma dinámica tendremos una barra de desplazamiento que nos permitirá establecer la velocidad con que se va a representar. El icono de las tijeras, situado en la parte derecha de la aplicación nos permitirá borrar los puntos dibujados anteriormente al pulsar del botón.

Destacar que los botones de desplazamiento y zoom, situados en la parte derecha de la aplicación, serán válidos tanto para la representación dinámica como para la estática.

El usuario tendrá la oportunidad, siempre que lo desee, de establecer el grosor de la representación gráfica. También podrá imprimir o guardar el panel de representación.

Si hemos decidido realizar la representación dinámica del muelle y el eje XT, no se podrá elegir la representación estática del movimiento.

El uso de los botones de acción será obligatorio para cualquier tipo de representación. El botón *play* será el encargado de iniciar la representación elegida, el pause detendrá la representación dinámica en el momento en que lo pulsemos y el botón *stop* limpiará el panel de representación.

3.4 Implementación

En este apartado, se explican los detalles más importantes referidos a la implementación del *Applet*. Explicaremos las clases utilizadas y los métodos más importantes.

3.4.1 AppletOscilador

Se trata de la clase principal de la aplicación. Como se puede comprobar a continuación hereda de la clase *JApplet*.

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

Cabe destacar de esta clase una función importante como es el *init()*:

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

Este método *init()* se encarga de realizar tareas de inicialización, por ejemplo, establecer las propiedades de los controles, disponerlos en el *Applet*, cargar imágenes, etc. El método *init()* se llama una sola vez, después el navegador llamará al método *paint*.

Los métodos de desplazamiento *(jButtonDer, jButtonIzq, jButtonArriba, jButtonAbajo y jButtonDesplDef)* permitirán al usuario poder desplazar la representación gráfica hacia las distintas direcciones posibles (izquierda, derecha, arriba, abajo) y situarla en su posición inicial. Todos estos métodos llamarán a la función actualizar y únicamente realizarán la modificación si la representación ha sido iniciada.

```
private void jButtonDerActionPerformed(java.awt.event.ActionEvent evt) {
    if (iniciado){
        desplHoriz = desplHoriz + 50;
        actualizar();
    }
}
```

Los métodos de zoom *(jButtonAcercar, jButtonAlejar y jButtonZoomDef)* permitirán al usuario poder acercar, alejar y establecer el zoom por defecto de la representación gráfica, siempre que se haya inicializado la aplicación. Estos métodos llamarán a la función *actualizar()* y únicamente realizará la modificación si la representación ha sido iniciada. Ejemplo del método *jButtonAcercar*.

```
private void jButtonAcercarActionPerformed(java.awt.event.ActionEvent evt) {
    if (iniciado){
        zoom = zoom * 1.1;
        actualizar();
    }
}
```

La función *leerParams()* se encargará de leer los parámetros y variables introducidos por el usuario para las dos gráficas. Además de esto, si el usuario introduce de manera incorrecta los parámetros/variables llamará al método *errorParametros()* que se encargará de mostrar un mensaje de error y poner los *jTextField* incorrectos de color rojo. Nota: es sólo un extracto del código.

```
private void leerParams(){
    try
    {
        // Gr□fica 1
        x = Double.parseDouble(jTextFieldx.getText());
        // Gr□fica 2
        x2 = Double.parseDouble(jTextFieldx2.getText());
        malParametros=false;
    }
    catch (Exception e)
    {
        malParametros=true;
        errorParametros();
        System.err.println("Exception caught: " + e.getMessage());
    }
}
```

Este método *(jPlay)* será el encargado resolver el oscilador según su tipo, establecer los datos de los hilos e iniciarlos.

```
private void jPlayActionPerformed(java.awt.event.ActionEvent evt) {
    leerParams();
    if (!malParametros){
        {...}
        g=jGrafica.getGraphics();

        Parametros param = new Parametros(x,y,w,μ,F,r,t,r1Graf1,r2Graf1,m1);
        Parametros param2 = new Parametros(x2,y2,w2,roz2,F2,r2,t2,r1Graf2,r2Graf2,m2);

        if (jComboBoxGrafica.getSelectedIndex()==0){
            OsciladorArmonico o = new OsciladorArmonico(h,w);
            OsciladorArmonico o2 = new OsciladorArmonico(h2,w2);
            l1 = o.resolverRunge(param);
            l2 = o2.resolverRunge(param2);
        }else if (jComboBoxGrafica.getSelectedIndex()==1){
            OsciladorAmortiguado oa = new OsciladorAmortiguado(h,w,μ);
            OsciladorAmortiguado oa2 = new OsciladorAmortiguado(h2,w2,roz2);
            l1 = oa.resolverRunge(param);
            l2 = oa2.resolverRunge(param2);
        }else if (jComboBoxGrafica.getSelectedIndex()==2){
            OsciladorForzado of = new OsciladorForzado(h,w,μ,F,r,t);
            OsciladorForzado of2 = new OsciladorForzado(h2,w2,roz2,F2,r2,t);
            l1 = of.resolverRunge(param);
            l2 = of2.resolverRunge(param2);
        }
        p.setDatos(jGrafica,g,l1,l2,param,param2,velocidad,
                colorGrafica1,colorGrafica2, zoom,desplHoriz,desplVert,jProgressBarTiempo,
                jSpinnerGrosor.getValue().toString(),true);

        p2.setDatos(jGrafica,g,l1,l2,param,param2,velocidad,
                colorGrafica1,colorGrafica2, zoom,desplHoriz,desplVert,jProgressBarTiempo,
                jSpinnerGrosor.getValue().toString(),false);

        {...}
        if (!pausa){
            hilo = new Thread(p);
            hilo.start();
            if (jRadioButtonRepDin.isSelected()){
                hilo2 = new Thread(p2);
                hilo2.start();
            }
        }
        if (pausa){
            pausa=!pausa;
        }
    }
}
```

El botón de parar *(jParar)* será el encargado de detener el hilo, deshabilitar las opciones de parar y pausar, limpiar el panel de representación y establecer la barra de progreso a 0. Un aspecto a destacar es que si el usuario eligió la opción de representación del muelle también detendrá el segundo hilo.

```
private void jPararActionPerformed(java.awt.event.ActionEvent evt) {
    pausa = false;
    hilo.stop();
    if (jRadioButtonRepDin.isSelected()){
        hilo2.stop();
    }
    jPlay.setEnabled(true);
    jParar.setEnabled(false);
    jPausar.setIcon(new javax.swing.ImageIcon
            (getClass().getResource("/proyosciladorarmonico/pause.png")));
    jPausar.setEnabled(false);
    jGrafica.getGraphics().clearRect(0,0,jGrafica.getWidth(),jGrafica.getHeight());
    jProgressBarTiempo.setValue(0);
    iniciado = false;
}
```

Con el método *jButtonImp* se podrá realizar la impresión del panel de representación mediante la impresora o por medio del programa *PDF Creator* (siempre que este instalado). Una vez pulsado el botón de impresión aparecerá un diálogo en el que el usuario establecerá las características con las que será impreso.

```
private void jButtonImpActionPerformed(java.awt.event.ActionEvent evt) {
        imprimir = true;
        File fichero = new File("captura.png");
        try {
            p.generarImagen(jGrafica, fichero);
        } catch (IOException ex) {
            Logger.getLogger(AppletOscilador.class.getName()).log(Level.SEVERE, null, ex);
        }

        // Archivo a imprimir
        String file = "captura.png";

        // Definimos el tipo a imprimir
        DocFlavor docFlavor = DocFlavor.INPUT_STREAM.PNG;

        // Establecemos algunos atributos de la impresora
        PrintRequestAttributeSet aset = new HashPrintRequestAttributeSet();
        aset.add(MediaSizeName.ISO_A4);
        aset.add(new Copies(1));

        Doc docPrint;
        try {
                docPrint = new SimpleDoc(new FileInputStream(file), docFlavor, null);
        } catch (FileNotFoundException e1) {
                e1.printStackTrace();
                return;
        }

        PrinterJob job;
        job = PrinterJob.getPrinterJob();

        // Diálogo para elegir el formato de impresión
        job.printDialog();
        PrintService printService = job.getPrintService();

        // Inicio el proceso de impresion...
        DocPrintJob printJob = printService.createPrintJob();
        try {
                printJob.print(docPrint, aset);
        } catch (PrintException e) {
                e.printStackTrace();
                return;
        }
        imprimir = false;
}
```

El método *jButtonGuardar* es el encargado de mostrar el selector de ficheros *(JFileChooser)*, responder a la opción elegida por el usuario y llamar al método encargado de generar la imagen *(generarImagen)*. Mediante el *jOptionPane* se devolverá un número entero que devolverá la opción elegida por el usuario. Si la variable selección es igual a 0 corresponderá a lo opción "Si" y si es igual a 1 corresponde a la opción "No". Cuando el usuario trate de guardar la imagen y el pintado haya sido iniciado entrará en el método *dialogExito* (perteneciente a la clase *OperacionesAuxiliares*). En cuanto al selector de ficheros, mencionar que el usuario podrá elegir la ruta que desee para guardar el archivo y sólo podrá guardar imágenes con formato *".png"*.

```
private void jButtonGuardarActionPerformed(java.awt.event.ActionEvent evt) {
    if (iniciado){
        guardar = true;
        int seleccion = JOptionPane.showOptionDialog(
            jButtonGuardar,
            "¿Desea guardar la imagen?",
            "Seleccione una opción",
            JOptionPane.YES_NO_CANCEL_OPTION,
            JOptionPane.QUESTION_MESSAGE,
            null,    // null para icono por defecto.
            new Object[] { "Si", "No"},   // null para YES, NO y CANCEL
            "opcion 1");

        if (seleccion == 0){
            JFileChooser fileChooser = new JFileChooser();
                        // Filtro para archivos .png
            FileNameExtensionFilter filtroImagen = new FileNameExtensionFilter("PNG","png");
            fileChooser.setFileFilter(filtroImagen);
            int selec = fileChooser.showSaveDialog(this);
            if (selec == JFileChooser.APPROVE_OPTION)
            {
                                // Establecemos la ruta y el formato
               File fichero = new File(fileChooser.getSelectedFile().getAbsolutePath()+".png");
                try {
                    p.generarImagen(jGrafica,fichero);
                } catch (IOException ex) {
                    Logger.getLogger(AppletOscilador.class.getName()).log(Level.SEVERE, null, ex);
                }
            opAux.dialogExito();
            }
        }
    }
}
```

Los siguientes métodos *(jRadioButtonEsp y jRadioButtonIng)* serán los encargados de cambiar el idioma de la aplicación, tanto para inglés como para castellano, y de establecer este componente como seleccionado.

```
private void jRadioButtonEspActionPerformed(java.awt.event.ActionEvent evt) {
    jRadioButtonEsp.setSelected(true);
    jRadioButtonIng.setSelected(false);
    jLabelTitulo.setText("Oscilador armónico");
    jRadioButtonEstatica.setText("Estática");
    {...}
}
private void jRadioButtonIngActionPerformed(java.awt.event.ActionEvent evt) {
    jRadioButtonIng.setSelected(true);
    jRadioButtonEsp.setSelected(false);
    jLabelTitulo.setText("Harmonic oscillator");
    jRadioButtonEstatica.setText("Static");
    {...}
}
```

Este método *(actualizar)*, como su propio nombre indica, se encargará de actualizar los parámetros y datos. Destacar que si la representación fue representada de forma estática, este método se encargará de lanzar otro nuevo hilo. Por ultimo, llamará al método *update(g)* que se encargará de limpiar el panel de representación.

```
private void actualizar() {
    modificado = true;
    Parametros param = new Parametros(x,y,w,μ,F,r,t,r1Graf1,r2Graf1,m1);

    Parametros param2 = new Parametros(x2,y2,w2,roz2,F2,r2,t2,r1Graf2,r2Graf2,m2);

    p.setDatos(jGrafica, g, l1, l2, param, param2,
            velocidad, colorGrafica1, colorGrafica2,
            zoom, desplHoriz, desplVert,jProgressBarTiempo,
            jSpinnerGrosor.getValue().toString(),true);

    p2.setDatos(jGrafica, g, l1, l2, param, param2,
            velocidad, colorGrafica1, colorGrafica2,
            zoom, desplHoriz, desplVert,jProgressBarTiempo,
            jSpinnerGrosor.getValue().toString(),false);

    if ((seleccionEstatico)||(fin)){
        hilo = new Thread(p);
        hilo.start();
    }
    if (!jRadioButtonDinamica.isSelected()){
        hilo2 = new Thread(p2);
        hilo2.start();
    }
    jGrafica.update(g);
}
```

3.4.2 PintarGrafica

Como se puede comprobar la clase *PintarGrafica* extiende de la interfaz *Runnable*. Esta interfaz nos permite soportar el subprocesamiento múltiple en una clase que ya extiende a otra clase distinta de *Thread*, ya que Java no permite que una clase extienda a más de una clase a la vez.

```
public class PintarGrafica implements Runnable{
```

Una vez dentro de *run()*, se pueden comenzar las sentencias de ejecución como en otros programas. Sirve como rutina *main()* para los hilos y cuando éste termina, también lo hace el hilo. Todo lo que se quiera que haga el hilo de ejecución ha de estar dentro de *run()*, por eso cuando se dice que un método es *Runnable*, es obligatorio escribir un método *run()*.

```
@Override
public void run() {
    comenzarDibujo(grap);
}
```

Mediante este método *(generarImagen)* se generará la imagen del panel de representación. Está clase recibirá el panel de representación y el fichero donde será guardado (como argumentos). Se encargará de pintar el oscilador elegido por el usuario en un archivo de imagen (en nuestro caso ”*.png*”). Para ello, se cogerá el contexto gráfico de la imagen y se pintará sobre ella llamando a la representación estática.

```
public void generarImagen(JPanel panelAux,File f) throws IOException{
    Graphics off = null;
    fich = f;
    im = (BufferedImage) panelAux.createImage(panelAux.getWidth(),panelAux.getHeight());
    off = im.getGraphics();
    if (!AppletOscilador.repDinam){
        if (!AppletOscilador.seleccionEstatico) {
            AppletOscilador.seleccionEstatico = true;
            this.pintarEstatico(off, l1, l2, params, params2, panelAux, progress);
            AppletOscilador.seleccionEstatico = false;
        }else{
            this.pintarEstatico(off, l1, l2, params, params2, panelAux, progress);
        }
        ImageIO.write(im, "png", fich);
    }else{
        this.pintarGraficaXTPequenaEst(off, params, progress);
        while (!continuar){
            System.out.println("Vacio");
        }
        this.muelleGuardar(off);
        ImageIO.write(im, "png", fich);
    }
    AppletOscilador.guardar = false;
}
```

El método *setDatos* es el encargado de recopilar datos y componentes como pueden ser el panel de representación, los gráficos, parámetros, color de la gráfica…

```
public void setDatos(JPanel p, Graphics g, ArrayList lista1, ArrayList lista2,
        Parametros par, Parametros par2,
        Color cgrafica1, Color cgrafica2,
        double z, int dX, int dY, JProgressBar prog, String gros, boolean mue){
    panel=p;
    grap=g;
    {...}
}
```

El siguiente método *paint* [7] será el encargado de llevar a cabo el pintado de los distintos tipos de osciladores. A la hora del pintado del muelle dinámico, se ha implementado la técnica del doble buffer, que se encarga de dibujar sobre un *BufferedImage*, evitando dibujar sobre la pantalla. Cuando el dibujo sobre el *BufferedImage* está completo, se pega directamente encima de la pantalla, evitando con esto el parpadeo.

```
public void paint(Graphics g){
    //INICIO DE DIBUJO
    if ((AppletOscilador.repDinam) & (!AppletOscilador.curvaRes)){
     try {
            if (tmuelle){
                ejes(g);
                pintarGraficaXTPequena(g,params,progress);
            }else{
                i++;
                g.drawImage(imag, 0, 0, width, height/2, panel);
                alargarMuelle(l1,l2,offScreen,i);
            }
        } catch (InterruptedException ex) {
            Logger.getLogger(PintarGrafica.class.getName()).log(Level.SEVERE, null, ex);
        }
    }else{
        if (AppletOscilador.seleccionEstatico){
            ejes(g);
            pintarEstatico(g,l1,l2,params,params2,panel,progress);
        }else{
            try {
                ejes(g);
                pintarDinamico(g,l1, l2, params, params2, panel,progress);
            } catch (InterruptedException ex) {
                Logger.getLogger(PintarGrafica.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Normalmente, lo primero que suele hacer *update* es borrar la ventana y luego llamar al método *paint*. La parte del código que borra la ventana es la que causa el

parpadeo, por lo tanto, hemos de redefinir *update* en la clase que describe el *Applet*. Si dibujamos en la ventana sin borrarla previamente eliminamos el parpadeo.

```
public void update(Graphics g){
    paint(g);
}
```

El método *comenzarDibujo* será el encargado de comenzar el pintado mediante la llamada a la función *repaint()*. Si el usuario decidió pintar la representación dinámica del muelle y el eje XT, el método se encargará de crear la imagen y pasarle el contexto gráfico de ésta.

```
public void comenzarDibujo(Graphics g){
    if ((AppletOscilador.repDinam) & (!AppletOscilador.curvaRes)){
        if (imagen == null || width != panel.getWidth() || height != panel.getHeight()) {
            width = panel.getWidth();
            height = panel.getHeight();
            imag = (BufferedImage) panel.createImage(width, height/2);
        }
        offScreen = (Graphics2D) imag.getGraphics();
    }
    repaint();
}
```

El método *PintarEstático* se encargará, como su propio nombre indica, de llevar a cabo la representación estática de los todos los tipos de osciladores y la curva de resonancia. Realizará la representación para el eje $X\dot{X}$, eje *XT*, y para eje *A/R* de la curva de resonancia. Para ello, hará llamadas a los métodos *pintarEstaticoXY*, *pintarEstaticoXT* y *pintarEstaticoResonancia*.

En cuanto a su desarrollo, obtendremos las variables necesarias para la representación del punto, calcularemos su posición y las dibujaremos sobre el contexto gráfico del panel de representación.

Cabe destacar de este método las variables *limitGuardar* y *limitCortar*. Estas dos variables globales serán calculadas en el método *pintarDinamico* y nos servirán para conocer el límite en el que se tiene que dar lugar al salvado y cortado de la imagen.

A continuación, se mostrará una parte de la implementación del método *pintarEstáticoXY*.

```
public void pintarEstaticoXY(Graphics g, ArrayList l1, ArrayList l2 {
    {...}
    if (((AppletOscilador.guardar) || (AppletOscilador.imprimir))&& (limitGuardar!=0)){
        size = limitGuardar;
    }else{
        size = l1.size();
    }

    for (int i=0; i<size; i++){
        p1 = (VarRunge) l1.get(i);
        x1 = p1.getx();
        y1 = p1.gety();

        p2 = (VarRunge) l2.get(i);
        x2 = p2.getx();
        y2 = p2.gety();

        posx1 = (int) ((x1* zoomPanel +1) * xCero);
        posy1 = (int) ((alto - ((y1 * zoomPanel+1) * yCero)));

        posx2 = (int) ((x2* zoomPanel+1) * xCero);
        posy2 = (int) ((alto - ((y2 * zoomPanel+1) * yCero)));

        if (AppletOscilador.grafica1){
            g.setColor(colGrafica1);
            if (i>=limitCortar) {
                g.drawLine((int)posantx1+desX, (int)posanty1+desY, (int)posx1+desX, (int)posy1+desY);
            }
        }

        if (AppletOscilador.grafica2){
            g.setColor(colGrafica2);
            if (i>=limitCortar) {
                g.drawLine((int)posantx2+desX, (int)posanty2+desY, (int)posx2+desX, (int)posy2+desY);
            }
        }

        posantx1 = posx1;
        posanty1 = posy1;

        posantx2 = posx2;
        posanty2 = posy2;
    }
}
```

El método *pintarDinámico*, se encargará de llevar a cabo la representación dinámica de todos los tipos de osciladores y de la curva de resonancia. Al igual que en la representacion estática, mediante este método se podrá realizar la representación para los ejes $X\dot{X}$, XT y A/R (curva de resonancia).

Hará una labor bastante parecida a la del pintado estático, salvo algunas diferencias que se mostrarán a continuación.

El método *sleep* provoca que el intérprete ponga al hilo en curso a dormir durante el número de milisegundos que se indiquen en el parámetro de invocación. Una vez transcurridos esos milisegundos, dicho hilo volverá a estar disponible para su ejecución.

```
Thread.sleep((long) java.lang.Math.abs(((101-AppletOscilador.velocidad)*0.1)));
```

Si el usuario decide pulsar el botón cortar, se realizará un limpiado del panel de representación y se establecerá el limite de cortado en caso, de que posteriormente se desee guardar la imagen.

```
if (AppletOscilador.cortarTramo==true){
    limitCortar = i;
    panel.getGraphics().clearRect(0, 0, panel.getWidth(), panel.getHeight());
    ejes(g);
    AppletOscilador.cortarTramo = false;
}
```

Destacar, que si el usuario decide realizar una pausa, el método entrará en un bucle while, que no hará nada hasta que el usuario decida detenerla y establecerá la variable *limitPintado* (para el guardado de la imagen).

```
if (AppletOscilador.pausa) {
    limitGuardar = i;
}

while (AppletOscilador.pausa){

}
```

Por último destacar, que en caso de que el usuario haya decido dar zoom, desplazamiento u otras operaciones de modificación, el método se encargará de repintar los puntos anteriores.

A continuación, se mostrará un extracto del método *pintarDinamicoXY*.

```
public void pintarDinamicoXY(Graphics g, ArrayList l1, ArrayList l2,
        JProgressBar prog) throws InterruptedException{
    {...}
    for (int i=0; i<l1.size(); i++){

        if ((AppletOscilador.guardar) || (AppletOscilador.imprimir)) {
            limitGuardar = i;
            break;
        }
        Thread.sleep((long) java.lang.Math.abs(((101-AppletOscilador.velocidad)*0.1)));
        {...}
        int resultProg = 100 * i / l1.size();
        prog.setValue(resultProg+1);

        if (AppletOscilador.cortarTramo==true){
            limitCortar = i;
            panel.getGraphics().clearRect(0, 0, panel.getWidth(), panel.getHeight());
            ejes(g);
            AppletOscilador.cortarTramo = false;
        }

        if (AppletOscilador.pausa) {
            limitGuardar = i;
        }

        while (AppletOscilador.pausa){

        }
        if (AppletOscilador.modificado){
            {...}
            for (int j=0; j<i; j++){
                p1Back = (VarRunge) l1.get(j);
                x1Back = p1Back.getx();
                y1Back = p1Back.gety();

                p2Back = (VarRunge) l2.get(j);
                x2Back = p2Back.getx();
                y2Back = p2Back.gety();

                posx1Back = (int) ((x1Back* zoomPanel +1) * ancho / 2);
                posy1Back = (int) ((alto - ((y1Back * zoomPanel+1) * alto / 2)));

                posx2Back = (int) ((x2Back* zoomPanel+1) * ancho / 2);
                posy2Back = (int) ((alto - ((y2Back * zoomPanel+1) * alto / 2)));

                if (AppletOscilador.grafica1){
                    g.setColor(colGrafica1);
                    g.drawLine((int)posantx1+desX, (int)posanty1+desY, (int)posx1Back+desX, (int)posy1Back+desY);
                }
                if (AppletOscilador.grafica2){
                    g.setColor(colGrafica2);
                    g.drawLine((int)posantx2+desX, (int)posanty2+desY, (int)posx2Back+desX, (int)posy2Back+desY);
                }

                posantx1 = posx1Back;
                posanty1 = posy1Back;
                posantx2 = posx2Back;
                posanty2 = posy2Back;
                while (AppletOscilador.pausa){

                }
            }
            AppletOscilador.modificado = false;
        }
        {...}
        if (AppletOscilador.pausa) {
            limitGuardar = i;
        }

        while (AppletOscilador.pausa){

        }
    }
    AppletOscilador.fin = true;
}
```

El siguiente método *alargarMuelle* será el encargado de representar dinámicamente el movimiento del muelle. Dado que, este método dibujará sobre el entorno gráfico de la imagen (técnica del *doble buffer*), lo primero que se hará, será el limpiado de la parte superior del panel y se pintarán los ejes del muelle. Una vez hecho esto, mediante las funciones *drawPolyline* y *fillRect* nos encargaremos del dibujado del movimiento del muelle. Estos métodos, contendrán las coordenadas a pintar y cambiarán por cada nueva interacción del muelle. Por último destacar que, por cada pintado del muelle se llamará a la función repaint, que se encargará de pintar el siguiente interacción.

```
private void alargarMuelle(ArrayList l1, ArrayList l2, Graphics g,
        int i) throws InterruptedException{

    {...}
    int [] xret = {100,
        (int)(posx1+diviMuelle),
        (int)(posx1+(diviMuelle*2)-retro),
        (int)(posx1+(diviMuelle*3)-retro2),
        (int)(posx1+(diviMuelle*4)-retro3),
        (int)(posx1+(diviMuelle*5)-retro4),
        (int)(posx1+(diviMuelle*6)-retro5),
        (int)(posx1+(diviMuelle*7)-retro5-50)};

    int [] yret = {yCero,yCero-25,yCero,yCero-25,yCero,yCero-25,
        yCero,
        yCero};

    synchronized(g){
        g.clearRect(0, 0, width, height/2);
        pintarCaja(g);
        g.setColor(Color.gray);
        // Pinta una secuencia de líneas conectadas
        g.drawPolyline(xret, yret, 8);
        g.setColor(colGrafica1);
        // Rellena el rectángulo especificado
        g.fillRect((int)(posx1+(diviMuelle*7)-retro5-50), yCero-25, 50, 50);
        g.setColor(Color.BLACK);
    }
    {...}
    if ((!AppletOscilador.fin) && (!AppletOscilador.guardar) && (!AppletOscilador.imprimir)) {
        repaint();
    }
    {...}
}
```

Los siguientes métodos se encargarán de representar los ejes para los distintos tipos de representación.

```
public void ejesxy(Graphics g){
public void ejesGraficaXT(Graphics g){
public void ejesResonancia(Graphics g){
public void ejesGraficaGrandeXT(Graphics g){
```

3.4.3 Runge

Esta clase será la encargada de realizar el método Runge-Kutta, que es un método de resolución de ecuaciones diferenciales.

Destacar de esta clase los siguientes aspectos:

- Se trata de una clase abstracta.

```
public abstract class Runge {
```

- Contiene los siguientes métodos abstractos:

```
abstract public double q1(double y);

abstract public double q2(double x, double y, double t);
```

- Calculará el número de iteraciones para el valor de *h =0.01*.

```
// Cálculo del número de iteraciones
Nit=(int) (100 / h);
```

- Llevamos a cabo el cálculo del *Runge-Kutta* de cuarto orden, establecemos las variables calculadas en la clase *VarRunge*, las añadimos al *ArrayList* y lo devolvemos.

```
for (j=0; j<Nit; j++){
    // 'x' e 'y' son las variables dependientes o
    // función y 't' es la variable independiente

    kx1=q1(y);
    ky1=q2(x,y,t);
    kx2=q1(y+h/2.*ky1);
    ky2=q2((x+h/2.*kx1),(y+h/2.*ky1),(t+h/2.));
    kx3=q1(y+h/2.*ky2);
    ky3=q2((x+h/2.*kx2),(y+h/2.*ky2),(t+h/2.));
    kx4=q1(y+h*ky3);
    ky4=q2((x+h*kx3),(y+h*ky3),(t+h));
    x=x+h/6.*(kx1+2.0*kx2+2.0*kx3+kx4);
    y=y+h/6.*(ky1+2.0*ky2+2.0*ky3+ky4);

    t=t+h;

    // Establecemos las variables calculadas en la clase VarRunge
    VarRunge p = new VarRunge(x,y,t);
    // Añadimos al ArrayList
    l.add(p);
}
return l;
```

3.4.4 Oscilador armónico

De esta clase hay que destacar los siguientes aspectos:

- Extiende de la clase *Runge*:

```
public class OsciladorArmonico extends Runge{
```

- Se declara el parámetro local ω que corresponde a la frecuencia angular.

```
// Frecuencia angular
protected double w;
```

- Esta clase contará con un constructor de dos parámetros (h y $\omega 0$). Éste se encargará de hacer una llamada a la superclase mediante la palabra reservada *super*, para obtener el valor de h, y se establecerá el valor de ω.

```
// Constructor
public OsciladorArmonico(double h, double w0){
    super(h);
    this.w=w0;
}
```

- Dispone de dos funciones heredadas de la clase *Runge* y llevarán a cabo los cálculos del oscilador armónico simple.

```
@Override
public double q1(double y) {
    return y;
}

@Override
public double q2(double x, double y, double t) {
    return -w*x;
}
```

3.4.5 Oscilador amortiguado

De esta clase hay que destacar los siguientes aspectos:

- Extiende de la clase *OsciladorArmonico*.

```
public class OsciladorAmortiguado extends OsciladorArmonico{
```

- Se declara el parámetro local μ que corresponde al coeficiente de rozamiento.

```
// Coeficiente de rozamiento
protected double μ;
```

- Esta clase contará con un constructor de tres parámetros (h, ω y μ). Éste se encargará de hacer una llamada a la superclase mediante la palabra reservada *super*, para obtener el valor de h y ω, y se establecerá el valor de μ.

```
public OsciladorAmortiguado(double h, double w, double μ){
    super(h, w);
    this.μ=μ;
}
```

- Dispone de dos funciones heredadas de la clase *Runge* y llevarán a cabo los cálculos del oscilador amortiguado.

```
public double q1(double y) {
    return y;
}

public double q2(double x, double y, double t) {
    return -w*x-μ*y;
}
```

3.4.6 Oscilador forzado

De esta clase hay que destacar los siguientes aspectos:

- Extiende de la clase *OsciladorAmortiguado*:

```
public class OsciladorForzado extends OsciladorAmortiguado{
```

- Se declaran los parámetros r (frecuencia de forzamiento externo) y F (fuerza de Newton).

```
// Frecuencia de forzamiento externo
protected double r;
// Fuerza de Newton
protected double f;
```

- Esta clase contará con un constructor de seis parámetros (h, w, μ, F, r *y* t). Éste se encargará de hacer una llamada a la superclase mediante la palabra reservada *super*, para obtener el valor de h, ω y μ, y se establecerá el valor de F *y* r.

```
// Constructor
public OsciladorForzado(double h, double w, double μ,
                        double f, double r, double t){
    super(h,w,μ);
    this.f=f;
    this.r=r;
}
```

- Dispone de dos funciones heredadas de la clase *Runge* y llevarán a cabo los cálculos de oscilador forzado.

```
public double q1(double y) {
    return y;
}

public double q2(double x, double y, double t) {
    return -w*x-μ*y+f*Math.cos(r*t);
}
```

3.4.7 Parametros

Clase que contendrá todas las variables y parámetros (excepto h) de los osciladores. Contiene dos constructores, uno será auxiliar y al otro se lo podrá pasar todos las variables y parámetros. Además, contará con todos los *get* y *set* correspondientes a cada parámetro.

A continuación, se mostrará el constructor por el cual se establecen los valores de las variables.

```
Parametros(double xAux, double yAux, double wAux,
        double rozAux, double FAux, double rAux, double tAux,
        double r1Aux, double r2Aux, double mAux){
    this.x = xAux;
    this.y = yAux;
    this.w = wAux;
    this.roz = rozAux;
    this.F = FAux;
    this.r = rAux;
    this.t = tAux;
    this.r1 = r1Aux;
    this.r2 = r2Aux;
    this.m = mAux;
}
```

3.4.8 OperacionesAuxiliares

En esta clase se encuentran los métodos auxiliares que servirán de gran ayuda a la hora de la realización de la aplicación. A continuación haremos mención a los siguientes.

La función *tamanio* nos devolverá el tamaño que ocupa la representación para el eje *XT*.

```
public int tamanio(int postMax, ArrayList l1, double zoomPanel,
        double tC, double d, int ale){
    int i=0, post1=0;
    double t1;
    while (post1<=postMax){
        if (post1>=postMax){
            break;
        }
        VarRunge punto1 = (VarRunge) l1.get(i);
        t1 = punto1.gett();
        post1 = (int) (((t1* zoomPanel +1) * tC)/ d +ale);

        i++;
    }
    return i;
}
```

El método *mensajeParam* devolverá el nombre de las/los variables/parámetros que han sido introducidas incorrectamente y pondrá de color rojo su correspondiente *jTextField*.

```
public String mensajeParam(JTextField txt, JLabel lbl, int numGraf, int numChar){
    String mensaje = "";
    String paco = "";
    paco = lbl.getText().substring(0, numChar);
    mensaje = mensaje+paco+" (Gráfica "+numGraf+"), ";
    txt.setBackground(Color.red);
    return mensaje;
}
```

3.5 Movimientos representativos del sistema

En el siguiente apartado se mostrarán los tipos de representaciones que puede realizar el usuario.

La *figura 32* mostrará una representación estática del oscilador armónico simple para una órbita mediante el tipo de eje $X\dot{X}$.

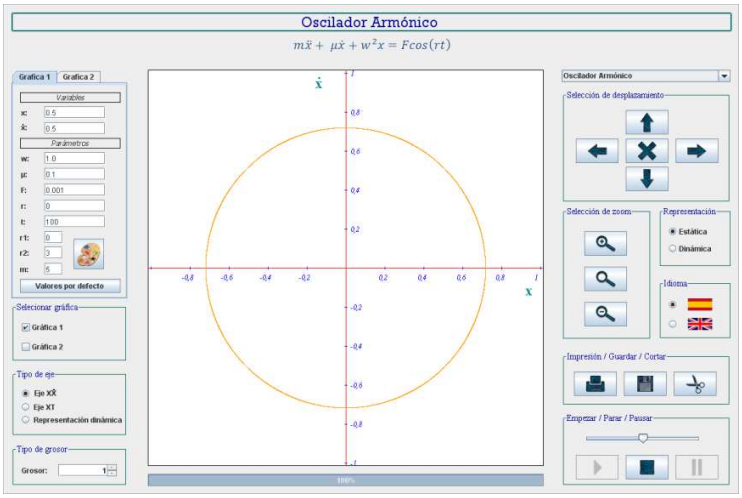

Figura 32. Representación estática del oscilador armónico para una órbita mediante el tipo de eje $X\dot{X}$.

La *figura 33* reflejará la representación dinámica del oscilador amortiguado para dos órbitas mediante el tipo de eje XT.

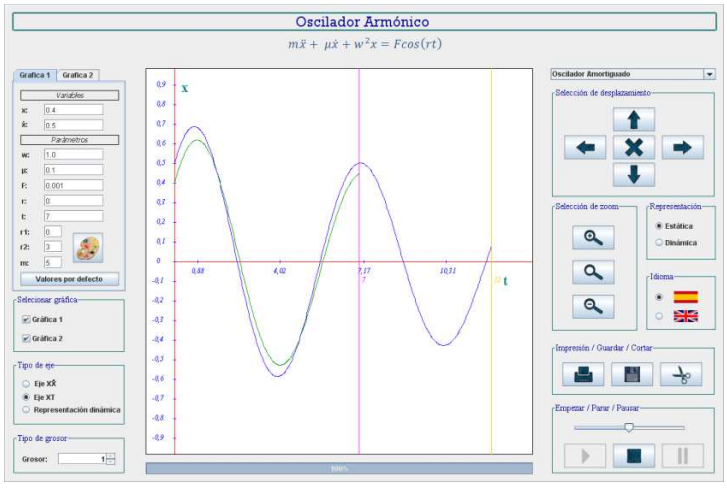

Figura 33. Representación dinámica del oscilador amortiguado para dos órbitas mediante el tipo de eje XT.

La *figura 34* mostrará la representación estática del oscilador forzado para una órbita y con mayor grosor mediante el tipo de eje $X\dot{X}$.

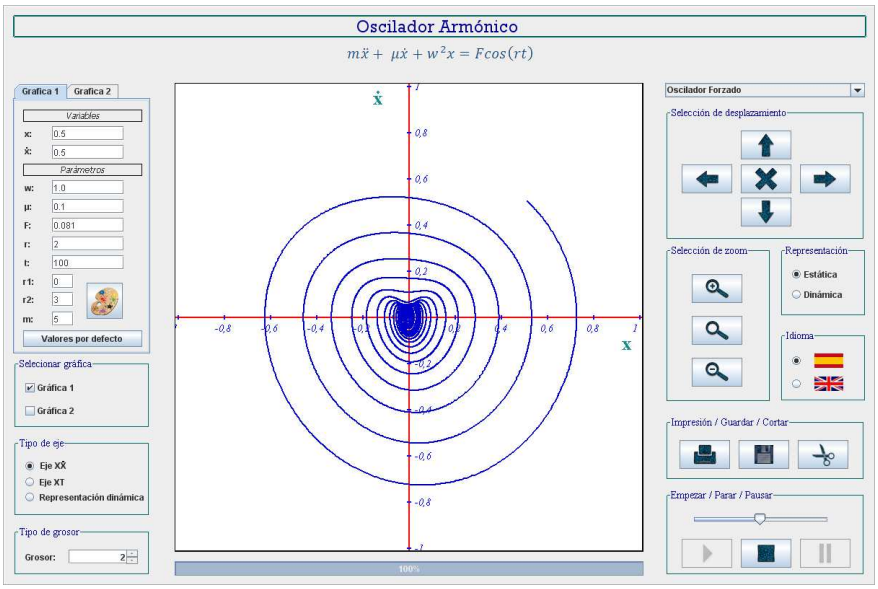

Figura 34. Representación estática del oscilador forzado para una órbita y con mayor grosor mediante el tipo de eje $X\dot{X}$.

La *figura 35* se podrá observar la representación dinámica del oscilador armónico simple mediante el muelle y el eje XT.

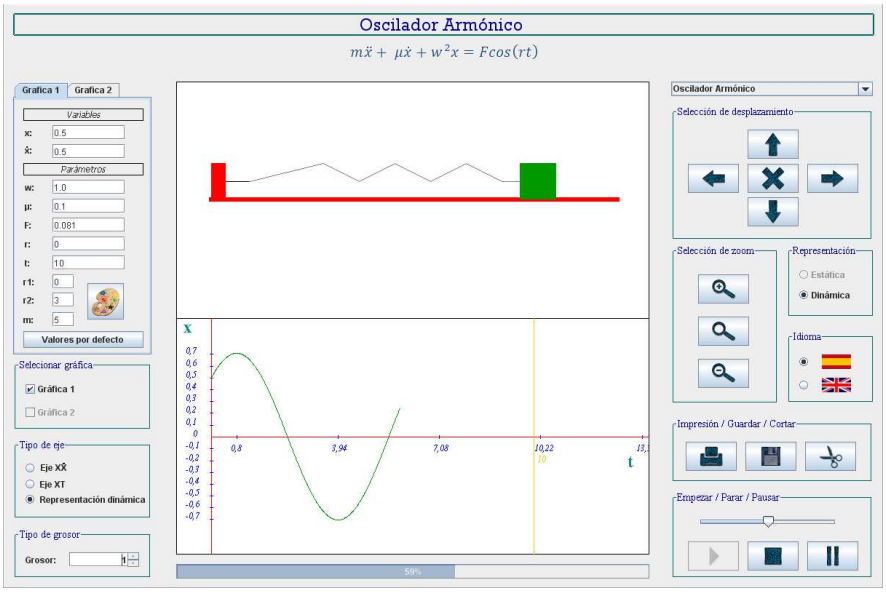

Figura 35. Representación dinámica del oscilador armónico simple mediante el muelle y el eje XT.

En la *figura 36* se mostrará la representación estática de la curva de resonancia para dos gráficas.

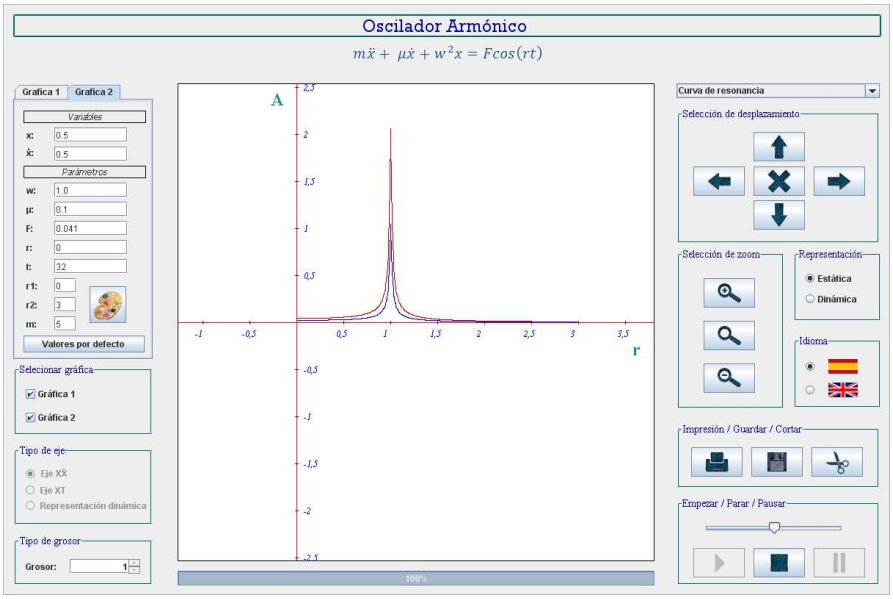

Figura 36. Representación estática de la curva de resonancia para dos gráficas.

En la *figura 37* se puede observar como aparece un diálogo de error indicando al usuario los parámetros/variables que ha introducido incorrectamente y la gráfica a la que pertenece.

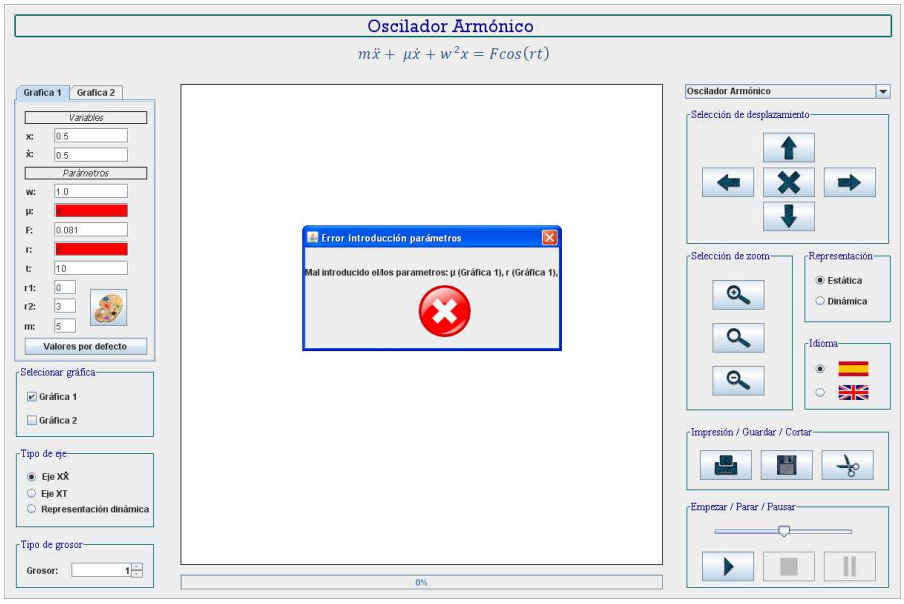

Figura 37. Error en la introducción de variables/parámetros.

3.6 Firmado del applet

El firmado del *Applet* es un aspecto muy importe a destacar, un *Applet* por defecto no puede acceder a los recursos del ordenador donde se esté visualizando, ni disco duro, impresora… El motivo es básicamente por seguridad, es decir, a la hora de visualizar un *Applet* no pueda, por poner un ejemplo, borrar los datos del disco duro. De todos modos, hay a veces que es necesario que pueda acceder a los recursos del ordenador, por lo tanto, hay que firmar digitalmente el *Applet*. Cuando el navegador visualiza un *Applet* firmado aparecerá una ventana con las siguientes características.

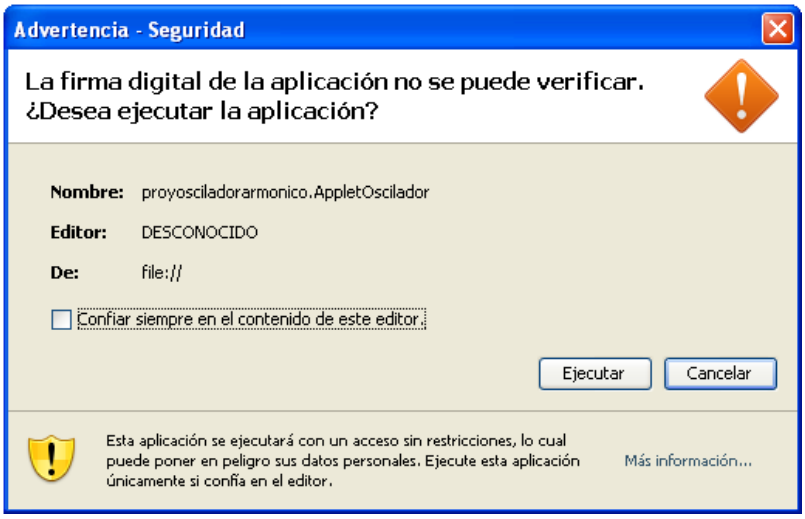

Figura 38. Advertencia de seguridad.

Si aceptamos, le comunicamos al navegador que confiamos en el creador de dicha aplicación, así que el navegador confiará en el *Applet* y le dará los permisos de acceso al disco duro, a la impresora, etc.

4. Conclusiones

En este apartado indicaremos todos los objetivos conseguidos a lo largo de la elaboración del proyecto.

- En primer lugar, indicar que se ha obtenido los conocimientos teóricos necesarios, en nuestro caso relacionado con el oscilador simple, amortiguado y forzado, para poder llevar a cabo una buena simulación del sistema físico.

- Hemos conocido los conceptos básicos de la programación orientada a objetos y nos hemos familiarizado con el entorno de programación *NetBeans*. En particular, se ha aprendido el desarrollo e implementación de los *Applets de Java*.
- En cuanto a la implementación, destacar que la implementación funciona correctamente y la aplicación permitirá obtener distintos tipos de representación. Además cumplirá todas las opciones de representación indicadas en el apartado de objetivos.
- Finalmente, la realización de la memoria contendrá toda la información del *Applet* desarrollado. Contendrá los objetivos a realizar, la descripción de los componentes, el desarrollo teórico del sistema simulado…

4.1 Logros alcanzados

El principal logro alcanzado ha sido incorporar los aspectos sugeridos por mis tutores en la aplicación, como aparece descrito a lo largo de este documento.

Este proyecto ha sido muy importante para mí para adentrarme en el mundo de las interfaces gráficas y entender el funcionamiento de un *Applet*. Me ha sido de gran ayuda para entender lo que son los hilos *(Threads)* y ponerlos en práctica. Un aspecto que me ha costado realizar ha sido la eliminación del parpadeo que producía el muelle a la hora del pintado.

4.2 Mejoras posibles

En toda aplicación siempre existe algún tipo de mejora y por esto en este apartado plantearé una serie de aspectos a tener en cuenta.

Por una parte, establecería una opción en el que al usuario se le permita seleccionar en el tipo de oscilador que prefiera y mostrarle una descripción teórica de éste. También ampliaría la opción de elección del idioma, estableciendo un listado con una cantidad amplia de ellos. Para el oscilador armónico simple, amortiguado y forzado, añadiría una gráfica que representara la energía total del móvil en función del tiempo.

5. Bibliografía

[1] Raymond A. Serway and John W. Jewett, Jr., *Física 1 3ª Edición,*

[2] William H.Press, Saul A. Teukolsky, William T. Vettering, Brian P. Flannery, *Numerical Recipes in C: The Art os Scentific Computing*, Cambridge University Press, 1992.

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

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

[5] Tutorial sobre applets en Java
http://docs.oracle.com/javase/tutorial/deployment/applet/

[6] Creación y control de hilos.
http://equis.umh.es/alex-bia/teaching/PC/material/hilos_tutorial-java/cap10-2.htm

[7] Información de la biblioteca utilizada para el pintado de las gráficas.
http://tabasco.torreingenieria.unam.mx/gch/Curso%20de%20Java%20CD/Documentos/froufe/parte15/cap15-2.html

[8] Tutoriales sobre Java Swing.
http://docs.oracle.com/javase/tutorial/uiswing/