Universidad Autónoma de Madrid

UAM

Integración de herramientas de simulación para la realización de aplicaciones de enseñanza

TESIS DOCTORAL

Escuela Técnica Superior de Informática

Marzo de 2000

Autor: Juan de Lara Jaramillo
Director: Manuel Alfonseca Moreno

Memoria presentada para optar al título de Doctor en Ingeniería Informática

Indice

Indice de figuras

■ Indice de tablas

■ Indice de listados

Agradecimientos

En primer lugar, tengo que agradecer a mi familia el apoyo y la ayuda que me han proporcionado durante todos estos años. Sin ellos terminar este trabajo habría sido imposible. A ellos, y muy especialmente a mi sobrina Irene, quiero dedicarles esta tesis.

Quiero dar las gracias muy especialmente a mi director de tesis, Manuel Alfonseca, por la cantidad de tiempo, paciencia y trabajo que ha tenido que emplear conmigo. Gracias por haber escuchado y encaminado mis no siempre afortunadas ideas. Haber trabajado con él ha sido, y espero que siga siendo, un verdadero placer y privilegio.

También tengo que mencionar y dar las gracias a la gente que ha colaborado y aportado ideas al presente trabajo, o bien han 'sufrido' alguna versión de *OOCSMP*, entre ellos Estrella Pulido, Federico García, Rosa Carro, Roberto Moriyón, Pablo Haya y Manuel Sánchez, de la Universidad de Murcia. En especial, la aportación de Estrella a los cursos de Internet, que se presentan en el capítulo X fue de gran valor. Muy agradables han sido también las largas charlas mantenidas con Alfonso. Mis más sinceros agradecimientos por su paciencia y sus interesantes comentarios. Tampoco puedo dejar de mencionar a todos los compañeros del departamento, y en especial a aquellos que fueron becarios en la época en que yo también lo fui. Mi gratitud a todos ellos. Muchas gracias también a Juana Calle, secretaria del departamento, por su amabilidad, paciencia y eficacia.

En otro orden de cosas, también quiero agradecer la paciencia que han tenido conmigo a mis compañeros Marcelo (espero que esto valga por una paella), Quique, José Julio y Chester (procuraré no tirarte más del bigote).

Mención aparte merecen las señoritas Caspita, Cunfutraquer y el señor Joselontxo, gracias a todos ellos/as por soportar mis palizas (si bien ellos también han contraatacado).

Espero no haberme dejado a nadie, en cualquier caso, inclúyase en este apartado aquél que lo considere oportuno.

Resumen

La simulación de sistemas [Mons97] es una de las ramas más antiguas de la informática. Estaba avanzada en los sesenta, y llegó a la madurez en los setenta. La simulación continua se ha programado tradicionalmente, bien en lenguajes de propósito especial, o bien de propósito general. Los lenguajes de simulación continua pueden ser de tres clases, dependiendo de su sintaxis: lenguajes de bloques [Alfo74], lenguajes orientados a las matemáticas y lenguajes de grafos [Karn90], [Lega80].

Muchos fenómenos físicos encontrados en ingeniería [Lapi82], biología [Brau93], biomecánica [Zinko96], física [Rubi98], mecánica de fluidos [Roach72], química [Pear93] y otros campos se pueden modelar por medio de ecuaciones diferenciales o en derivadas parciales. Normalmente el problema es demasiado complicado para resolverlo por medio de métodos analíticos, se necesita un enfoque numérico.

La programación orientada a objetos surgió en los sesenta, con un lenguaje de simulación discreta, el *SIMULA67* [Dahl66]. Este lenguaje ya incorporaba muchas de las ideas introducidas más tarde por Alan Kay en el primer lenguaje orientado a objetos de propósito general, Smalltalk [Digi90], y por Bjarne Stroupstrup en *C++* [Stro91].

Internet se está convirtiendo en una herramienta muy importante en la educación. Cada día hay más cursos basados en su uso [Thom97], [GNA97], que van desde una simple traducción de apuntes de clase, a la inclusión de elementos más sofisticados, como simulaciones, gráficos animados, etc. En particular, el lenguaje Java [Java99] ha hecho estos cursos más interactivos, de ejecución más rápida y más fácilmente transportables. El claro interés en este campo, ha creado una necesidad de herramientas adecuadas que ayuden en la elaboración de cursos, que deben hacer posible expresar todas las posibilidades ofrecidas por la enseñanza en Internet [Avia97], [Schu97].

Hemos desarrollado herramientas de simulación avanzadas que simplifican la generación de cursos educativos para Internet. El lenguaje que usamos es una extensión del antiguo *CSMP* (Continuous System Modelling Program) de *IBM* [IBM72]. Hemos llamado *OOCSMP* [Alfo97] al nuevo lenguaje, porque una de sus principales diferencias con *CSMP* es que le hemos añadido extensiones que le hacen orientado a objetos. Estas construcciones permiten simular mucho más fácilmente sistemas complejos basados en la interacción mutua de muchos agentes similares (que pueden ser modelados como colecciones de objetos).

OOCSMP también se ha extendido para resolver ecuaciones en derivadas parciales de segundo orden, en dos dimensiones, pudiendo realizar además análisis transitorio de las mismas. También se han añadido otras extensiones al lenguaje, tales como:

- Un álgebra de vectores y matrices, bucles implícitos sobre ellos, así como sobrecarga de los bloques del lenguaje, para que admitan escalares, vectores o matrices como parámetros.
- Posibilidad de añadir/borrar objetos a la simulación en tiempo de ejecución.
- Instrucciones para diseñar simulaciones alternativas a la simulación principal, disponibles desde un mismo programa.
- Instrucciones para el manejo de eventos discretos, similares a las construcciones *SI...ENTONCES*, y a *SI...ENTONCES...SI NO*.
- Incorporación de parámetros externos a la simulación.
- Primitivas para el uso de elementos multimedia y su sincronización con la ejecución de la simulación.
- Instrucciones que hacen posible la construcción de simulaciones distribuidas, mediante un esquema de distribución distinto de los que se han usado tradicionalmente en la práctica de la simulación distribuida.
- Nuevas salidas gráficas, pudiéndose combinar varias de ellas en la misma simulación. Un generador gráfico de mallas y dominios para la resolución de ecuaciones en derivadas parciales.
- Ampliación de las instrucciones de comentario con comentarios especiales, que ayudan a la generación de ficheros de documentación del modelo de forma automática, en formato *HTML*.
- Instrucciones para la configuración de la apariencia de la página *HTML* donde se incluirá el modelo de simulación. Estas instrucciones forman una capa del lenguaje de más alto nivel que las instrucciones de descripción de los modelos de simulación. Hemos llamado *SODA*

(Simulation COurse Description LAnguage) a estas instrucciones. De esta forma se facilita la construcción de cursos para Internet.

Se ha construido un compilador en *C++*, llamado *C-OOL* (a Compiler for the OOCSMP Language), que puede generar código *C++* o bien Java y/o páginas *HTML*. En el caso de que se genere *C++*, se puede optar también por usar las bibliotecas Amulet [Myer97], [Open97]. Si se elige esta última opción, es posible generar un fichero con los modelos de tareas de la interfaz [Alfo98], de forma que *ATOMS* [Garc98] pueda ofrecer servicios adicionales en tiempo de ejecución.

C-OOL genera una interfaz de usuario totalmente configurable mediante el uso de opciones de compilación. La interfaz permite la simulación interactiva y visual y la exploración del problema, de forma que se pueden cambiar valores de variables, la forma de los dominios de las *PDEs*, el intervalo elemental de tiempo, etc. y luego continuar la simulación. Es decir, permite responder a preguntas del tipo *"¿qué pasaría si...?"*. En el sistema *CSMP* anterior, esto no era posible, porque se mostraba sólo la salida final de la simulación.

Además *C-OOL* es capaz de compilar los antiguos modelos *CSMP*, manteniendo las ventajas expuestas anteriormente. Para compatibilidad con el mayor número posible de navegadores de Internet, y abarcar la mayor parte posible de versiones de estos, se ha optado por generar por defecto código para el *jdk1.1.0*, con el modelo antiguo de eventos, aunque es posible, mediante un flag del compilador, generar código Java para el nuevo modelo.

Para la validación del lenguaje, se han generado automáticamente varios cursos para Internet:

- Ecología.
- Gravitación.
- Electrónica básica.
- Ecuaciones en derivadas parciales y generación de mallas.

Todos estos cursos están accesibles desde: *http://www.ii.uam.es/~jlara/investigacion*.

El lenguaje *OOCSMP*, el compilador *C-OOL* y los cursos generados, se describen con detalle en el presente trabajo, así como numerosos ejemplos de uso y aplicación del lenguaje y el compilador.

Palabras clave: Simulación continua, Simulación basada en la web, Simulación enriquecida mediante multimedia, Simulación distribuida, generación de código, resolución numérica de ecuaciones en derivadas parciales, generación de mallas, visualización de datos en Internet, generación automática de cursos.

I. Introducción

I.1 Motivación

La orientación a objetos ha demostrado adaptarse adecuadamente al desarrollo de sistemas formados por diversos subsistemas que interaccionan entre sí. Con este paradigma, cada subsistema podría modelarse como un objeto o un *"framework"* de objetos que colaboran entre ellos. Se obtienen todavía más ventajas si existen en el sistema varios elementos con características similares, ya que estos podrían considerarse como objetos de la misma clase.

Recientemente, la orientación a objetos ha empezado a añadirse a lenguajes de simulación digital continua, tales como el intento de estandarización del grupo de trabajo sobre el lenguaje Modelica [Elmq97], [Mode99], con el que nuestro equipo ha colaborado. Hemos querido aprovechar estas ventajas de la orientación a objetos en un lenguaje de simulación, porque de esta forma nos es fácil modelar sistemas compuestos por elementos parecidos. Por ejemplo:

- Ecosistemas, formados por diversas especies.
- El sistema solar, que está compuesto por planetas.
- Un circuito electrónico, que está compuesto por módulos lógicos, etc.

Estos elementos similares se pueden encapsular como objetos de la simulación. De esta forma obtenemos además otras ventajas, como la posibilidad de reutilización de clases, el encapsulamiento o la herencia.

El rápido desarrollo de Internet, que se ha convertido en una herramienta clave en el campo educativo, nos sugirió la idea de construcción de herramientas que facilitasen el desarrollo de cursos para Internet basados en simulación. Los sistemas de simulación pueden interaccionar con la Web de distintas formas [Fish96]:

- Construyendo y ejecutando modelos distribuidos.
- Proveyendo herramientas para la Web, como intérpretes de lenguajes de simulación, o plug-ins [Schm99a].
- Integrando en cursos educativos simulaciones generadas previamente.

Nuestras opciones han sido la primera y la tercera. Para ello, se pensó en la posibilidad de que nuestro sistema generase *applets* y programas Java, así como páginas *HTML*, en las que estuviera incluido el applet de simulación. Estas páginas *HTML* se generan mediante instrucciones de más alto nivel (nivel *SODA*), desde las cuales, una simulación es un elemento hipermedia más, que se puede organizar dentro de una página de un curso. Además desde esta capa de alto nivel, se puede acceder a información del modelo de simulación.

Como alternativa, podría haberse elegido como lenguaje objeto cualquier otro lenguaje accesible mediante *CGI* (Common Gateway Interface). Otra opción distinta a un lenguaje de simulación, podría haber sido simplemente proporcionar un *framework* al usuario para construir el modelo. En este caso, el constructor del modelo habría tenido que programar en un lenguaje de propósito general, como *C++*, [Jone96], [Schw95] o Jaya [Heal97], [Page97], [Howe98], [Page98]. Las desventajas del uso de un lenguaje de propósito general para la simulación se expresan en el capítulo II. Hemos intentado paliar las desventajas que podría tener un lenguaje de simulación específico respecto a uno de propósito general, haciendo que, tanto la sintaxis del lenguaje, como los formatos de salida gráfica y la interfaz de usuario generada por el compilador, sean lo más flexibles posible. Para ello, se ha intentado que:

- Los formatos de salida sean flexibles, para que permitan expresar convenientemente los resultados de la simulación.

- La interfaz sea flexible (por ejemplo, configurable mediante opciones de compilación) y que permita explorar e interaccionar con el problema (por ejemplo, permitiendo cambiar variables o añadir objetos durante la simulación).

La integración de herramientas de simulación con elementos multimedia permite expresar de forma más rica el conocimiento que pretendemos comunicar, así como una mejor comprensión del problema por parte del alumno, ya que las explicaciones mediante vídeos, imágenes, textos, etc., no se realizan de forma estática en las páginas *HTML*, sino en los momentos adecuados durante la simulación.

La idea de extender el lenguaje para que sea capaz de resolver ecuaciones en derivadas parciales (*PDEs*), surgió debido a que gran parte de los fenómenos naturales son gobernados por ecuaciones de este tipo. Es de hacer notar la inexistencia hasta ahora de simuladores que sean capaces de combinar la simulación orientada a objetos con la resolución de ecuaciones en derivadas parciales.

También se han desarrollado bibliotecas para la generación de mallas (tres formas de generación distintas, estructuradas y no estructuradas), debido a que no es posible resolver una *PDE* si no se ha discretizado previamente el dominio sobre el que se quiere resolver. No se han encontrado generadores que pudieran ser utilizados por *OOCSMP*, debido a que ninguno podía ser llamado a través de Internet [NMGS99], [GID99]. Por ello se decidió construir bibliotecas (Java y C++) para la generación de mallas. También se planeó construir una interfaz gráfica, en lenguaje Java, para tener acceso al generador de mallas en tiempo de ejecución. Esto permite cambiar mallas, dominios, etc., mientras la simulación está en marcha.

Se intenta no limitarse a aplicaciones para la enseñanza, sino que las simulaciones pueden integrarse en otro tipo de aplicaciones que requieran un mayor rendimiento. Para ello se decidió que:

- Que el compilador pudiese generar un lenguaje más eficiente, tal como *C++*.
- Que el compilador pudiera generar código distribuido, para alcanzar mayor eficiencia en cierto tipo de simulaciones. Se aprovecha el paradigma de orientación a objetos para el desarrollo de un esquema de distribución original.

Por último, también es bueno que los modelos de simulación sean fáciles de mantener y de documentar. Para ello hemos construido un generador automático de documentación, que utiliza la información de la tabla de símbolos del compilador para generar la documentación en formato *HTML*.

El presente trabajo de investigación ha dado lugar a las siguientes publicaciones: 2 artículos en revistas internacionales (SIMULATION y APL Quote Quad, de *ACM*), 13 comunicaciones en congresos internacionales (ESS'97, ESS'98 y ESS'99, SESP'98, APL'98, ESM'98, EUROSIM'98, TOOLS EE'99, ASOO'99, EUROMEDIA'99 y EUROMEDIA'2000 e IMACS'2000 (dos artículos)), un artículo en revista nacional (ADIE) y dos comunicaciones en congresos nacionales (TAEE'98 y CONIED'99).

I.2 Plan de la tesis

Esta tesis se ha estructurado en 10 capítulos. El primero es esta introducción. El capítulo II se dedica a una breve introducción sobre la simulación digital continua y sobre la simulación basada en la Web.

En el capítulo III, se hace un pequeño resumen de las características y limitaciones del lenguaje *CSMP*, predecesor de *OOCSMP* (lenguaje que se ha desarrollado en el presente trabajo), así como de otros lenguajes (*NMF* y Modelica, dos intentos de estandarización de los lenguajes de simulación) y sistemas de simulación actuales (*ASCEND*, *AME*, *ECOSIM*, *Matlab/Simulink*, *STELLA* y otros).

En el capítulo IV se presenta una introducción a la tecnología de objetos, las extensiones orientadas a objetos que hemos incorporado a *OOCSMP*, así como otras extensiones, como la sobrecarga de bloques, la introducción de funciones y procedimientos, el álgebra de vectores y matrices, las construcciones para el manejo de eventos discretos, etc.

El capítulo V se dedica a las extensiones para la resolución de *PDEs*, y la definición de dominios y mallas. También se presenta nuestra herramienta *MGEN*, un entorno gráfico desarrollado en Java para el diseño de dominios y mallas. Por último se presentan algunos ejemplos de resolución de *PDEs* mediante distintos métodos, así como de uso de *MGEN*.

Los distintos tipos de salidas gráficas incorporadas al lenguaje se explican en el capítulo VI, así como las extensiones para multimedia. Las extensiones para la distribución se detallan en el capítulo VII.

La generación de las páginas *HTML* que dan lugar a los cursos de simulación y la generación automática de documentación se presentan en el apartado VIII. El entorno de desarrollo *C-OOL*, así como detalles sobre las interfaces que genera automáticamente, se presenta en el capítulo IX.

El capítulo X describe una serie de cursos generados mediante estas herramientas: gravitación, ecología, circuitos electrónicos y ecuaciones en derivadas parciales.

Por último el capítulo XI expone las conclusiones, limitaciones y líneas abiertas del presente trabajo.

II. Simulación

En la sección primera de este capítulo se presenta una breve introducción a la simulación digital continua y a los formalismos más usados para la descripción de modelos.

En la sección 2, se introduce el concepto de simulación basada en la Web (*Web-based simulation*) y el aprendizaje a distancia.

II.1 Simulación digital continua

La simulación digital continua consiste en el uso del computador digital para simular sistemas dinámicos. Un *sistema* es un conjunto de fenómenos observables, al que hemos separado del resto de universo para su estudio. Todo lo que no es el sistema, se llama *entorno* o *ambiente*. Un sistema interacciona con el entorno de dos formas: recibiendo de él influencias (*entradas* o *estímulos*) o actuando sobre él (*salidas* o *respuestas*).

El comportamiento de un sistema se rige por leyes físicas, que normalmente pueden ser expresadas por ecuaciones diferenciales o en derivadas parciales, quizá con alguna restricción de tipo-algebraico. Aunque el comportamiento de un sistema dinámico es continuo en el tiempo, en simulación digital continua, el tiempo se divide en intervalos discretos. Así pues, aunque el tiempo es la variable fundamental de la simulación, se pueden distinguir otros cuatro grupos de variables [Alfo99e]:

- **Exógenas:** su valor se origina fuera del sistema. Son las variables independientes y las de entrada.
- **De estado:** su valor define la situación actual del sistema.
- **Endógenas:** son variables auxiliares, resultado de la interacción entre las variables de los dos tipos anteriores.
- **De respuesta:** son las variables de salida, que pueden pertenecer a uno de los dos tipos anteriores.

Normalmente no se simula todo el sistema, sino sólo ciertos fenómenos que nos son de utilidad. Este proceso de reducir la complejidad del sistema, eliminando lo innecesario, se llama *abstracción*. Mediante la abstracción se obtiene un modelo simplificado del sistema [Fish88]. Suele ser útil considerar un sistema complejo como compuesto de un conjunto de entidades, que se modelan por separado. Una aproximación jerárquica tiende a reducir la complejidad y facilita el uso de distintos niveles de abstracción [Zeig76]. La modularidad ayuda a reducir la complejidad y facilita las modificaciones.

Un *modelo* es una descripción de un sistema mediante un lenguaje simbólico o una teoría [Mons97]. El modelado está orientado al usuario, según Minsky [Mins85] *"un objeto A es un modelo de un objeto B si un observador puede usar A para responder preguntas que le interesan acerca de B"*. Además, el modelo debe ser [deCo97]:

- Robusto. No debe haber situaciones dentro de sus límites de definición y suposiciones que lleven a resultados inconsistentes.
- Predictivo. Debe ser capaz de anticipar resultados que puedan ser verificados posteriormente.

La simulación en el computador nos es útil cuando no se puede experimentar directamente con el prototipo, es demasiado costoso, o poco ético. Además, es más fácil experimentar con el modelo simplificado que obtenemos cuando simulamos mediante el ordenador. La simulación ayuda a entender y solucionar problemas mediante un enfoque exploratorio, haciendo preguntas y observando los resultados [Pidd93]. En muchos casos, no es posible realizar experimentos con el sistema real, ya que cualquier prueba desafortunada podría llevar a situaciones de desastre. Con la simulación existe además la posibilidad de repetir experimentos con exactamente las mismas condiciones iniciales, cosa que es difícil al experimentar con ciertos sistemas reales. Uno de los objetivos de la simulación es estimar el efecto de condiciones extremas, hacer esto en la vida real puede ser peligroso o incluso ilegal en algunos casos.

Otra ventaja de la simulación digital, es que una vez obtenido el modelo de comportamiento del sistema, ese mismo modelo, nos puede servir para describir el comportamiento de otros sistemas totalmente distintos. Por ejemplo, la ecuación de ondas en tres dimensiones [Stra92], describe el comportamiento de las vibraciones en un sólido elástico, las ondas del sonido en el aire, ondas electromagnéticas (luz, radar, etc.), seismos, etc.

La figura II.1 muestra un esquema de los pasos necesarios hasta llegar a la simulación en el computador.

Idealmente, el *modelo base*, es un modelo que es válido para todos los experimentos posibles. Sin embargo, no es posible conocer totalmente el modelo base, sólo ciertas partes de su descripción. Se puede reducir la complejidad del modelo base estableciendo un marco experimental. Una vez establecido este, se puede construir un modelo simplificado válido dentro de ese marco.

Es necesario un uso cuidadoso de la simulación, ya que, debido a que la experimentación es fácil, el usuario puede perder la noción de lo que está haciendo, perdiendo el experimento todo el sentido. Mientras que técnicas analíticas [Chri83] (si se pueden aplicar), nos dan una explicación del comportamiento del sistema bajo condiciones arbitrarias, las ejecuciones de la simulación son útiles sólo cuando se aplican las condiciones experimentales.

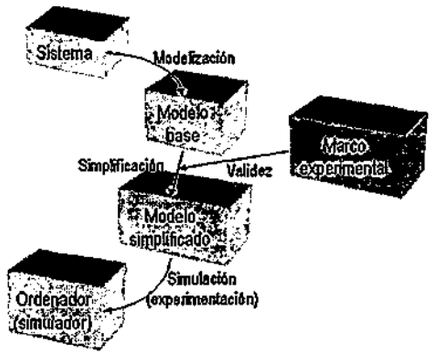


Figura II.1: Pasos de la simulación.

En la mayoría de los casos, no es posible un enfoque analítico, porque no se sabe plantear las ecuaciones con la precisión requerida, o bien resolverlas. Se hace necesario un enfoque numérico o el uso de la Simulación para resolver el problema [Pool77].

Por ejemplo, supongamos que queremos simular un peso que cuelga de un muelle [Stau96]. Podemos hacer abstracción del sistema físico, y hacer una primera simplificación, ya que quizá sólo nos interesen magnitudes tales como la posición, la velocidad y la energía del peso. Si despreciamos al rozamiento, el modelo puede quedar aún más simplificado, como puede observarse en la figura II.2.

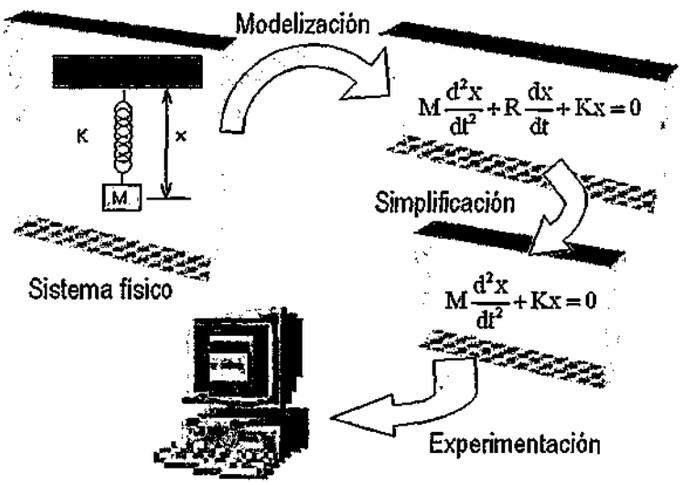


Figura II.2: Pasos en la simulación de un muelle.

El modelo inicial debe ser validado mediante la experimentación, comprobando con qué exactitud se adaptan las magnitudes que nos interesan a las del modelo real. Un modelo no puede utilizarse para realizar predicciones sobre el sistema real si no ha sido validado previamente con un número suficiente de casos reales. Cuando hay discrepancias, el modelo debe ser, bien refinado (por ejemplo, aumentando su

orden), o bien adaptado (por ejemplo, modificando parámetros) [Most97]. Esto da lugar a un ciclo en la simulación, que se muestra en la figura II.3.

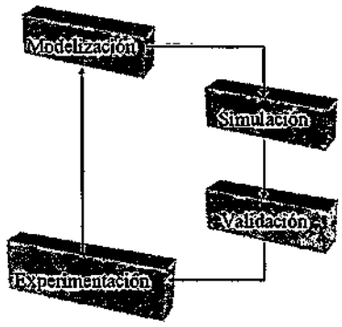


Figura II.3: Ciclo de la simulación.

La validación de un modelo se puede llevar a cabo de dos formas [Pidd93]:

- Validez de 'caja negra': Ignorando los detalles del funcionamiento interno del modelo, hay que comprobar si la salida que proporciona refleja el comportamiento del sistema real.
- Validez de 'caja blanca': Consiste en comprobar si cada componente del modelo refleja comportamientos conocidos y/o teorías válidas.

Poder experimentar con la simulación es la razón por la que se ha construido el modelo. Se pueden llevar a cabo dos tipos de experimentación:

- Clásica, en la que el modelo se ejecuta, y se obtienen unos resultados, que son analizados.
- Interactiva, en la ejecución es posible interaccionar con el modelo. Para ello, el programa de simulación debe permitir al usuario alguna forma de parar la simulación, para que pueda cambiar parámetros, y luego continuar. Además, durante la ejecución debe ser posible ir observando la dinámica del modelo mediante representaciones gráficas adecuadas. La simulación interactiva visual (*VIS*, Visual Interactive Simulation) y el Modelado Interactivo Visual (*VIM*, Visual Interactive Modelling) son hoy en día la norma en los sistemas de simulación. Facilitan la experimentación y la validación al usuario.

En un ordenador, la simulación se puede realizar de tres formas:

- **Simulación analógica:** El modelo está formado por circuitos electrónicos; el tiempo varía de forma continua, paralelamente entre el sistema y el modelo, con la posible excepción de un cambio de escala. El dispositivo sobre el que se ejecuta el modelo se llama *computadora analógica* o *analizador diferencial* y fue inventado en los años treinta por Vannevar Bush (1890-1974), aunque hoy día se encuentra en desuso, pues ha sido sustituido por las computadoras digitales, en las que se ejecutan los otros dos tipos de simulación.

- **Simulación digital continua:** el modelo matemático está formado por un conjunto de ecuaciones diferenciales o diferenciales-algebraicas (*DAE*) que se resuelven en una computadora digital; el tiempo es seudo-continuo y viene representado por un conjunto de valores predecibles, usualmente un muestreo con intervalo fijo (o cuasi-fijo), el *intervalo elemental.*

- **Simulación digital discreta:** el modelo matemático está formado por procesos estadísticos que se resuelven en una computadora digital; el tiempo es discreto y viene representado por una serie de valores no predecibles (muestreo aleatorio). El estado del sistema cambia cuando se produce un evento. Se puede obtener una definición dinámica del sistema avanzando el tiempo simulado desde un evento al siguiente. Este enfoque se denomina de *"siguiente evento"*[Mons97]. Los modelos discretos se suelen expresar en un formalismo llamado Especificación de Sistemas de Eventos discretos (*DEVS*) [Törn81].

Para la implementación de una simulación de tipo continuo en el ordenador, se hace necesario un lenguaje de programación adecuado, porque un lenguaje de propósito general, podría hacer demasiado

costoso tanto el desarrollo de simulaciones, como el ciclo de la simulación de la figura II.3. Además, un lenguaje de propósito especial, permite analizar, diseñar y entender mejor los sistemas físicos.

Existe un comité para la estandarización de los lenguajes de simulación continua (Continuous System Simulation Language, *CSSL*) [Cell86] que establece recomendaciones basadas en el enfoque de la Teoría de Sistemas [Spri82]. Este enfoque implica que se puede describir el sistema en términos de Entrada, Proceso y Salida. La base de este estándar es el hecho de que la ejecución de la simulación es el único experimento que se va a hacer con la simulación.

Las especificaciones *CSSL* de tipo estructural se muestran en la figura II.4. La sección de inicialización se ejecuta al principio de cada ejecución de la simulación. Suele dar valor a condiciones iniciales y calcular expresiones constantes. La sección dinámica contiene métodos y bloques funcionales que resuelven las ecuaciones diferenciales. La fase terminal se ejecuta al final de cada simulación.

Normalmente, en los lenguajes tipo *CSSL*, hay un algoritmo que ordena las ecuaciones en tiempo de compilación, para que se ejecuten en el orden apropiado.

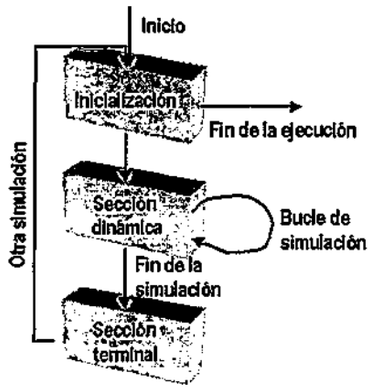


Figura II.4: Estructura de un lenguaje *CSSL*.

La figura II.5 muestra esquemáticamente los distintos tipos de formalismos para la descripción de sistemas físicos.

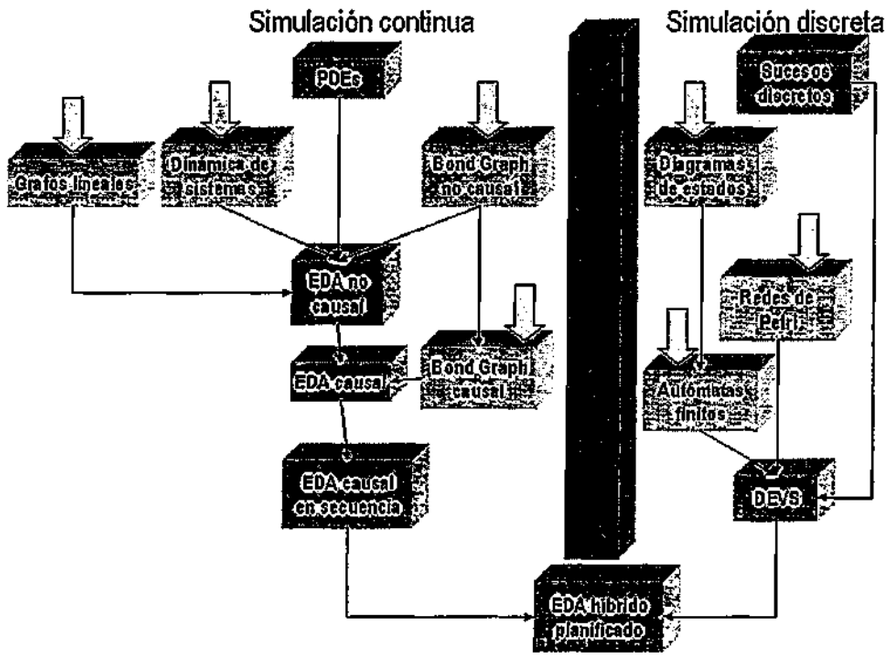


Figura II.5: Formalismos para la descripción de sistemas físicos.

Donde *EDA* significa ecuaciones diferenciales-algebraicas. Las flechas de mayor grosor sobre las cajas indican que se trata de un formalismo gráfico.

A continuación se detallan algunos de los formalismos de la simulación continua que aparecen en el gráfico anterior.

- **Diagramas de bloques:** Se basa en la interconexión de bloques. Cada bloque es un elemento electrónico que realiza ciertas operaciones (integración, sumatorio, etc.) sobre variables continuas. Simulan la programación con los antiguos computadores analógicos [Alfo71]. Los diagramas de bloques expresan las relaciones causa/efecto de las entradas y salidas de un sistema [Rosk72]. Ejemplos de sistemas que usan diagramas de bloques como formato de entrada son *SIAL/71* [Alfo74], *CSSP* [Ambo99] y EcoSim [Ecos99].

- **Bond graphs:** Este formalismo surgió a finales de los cincuenta, promovido por Paynter [Payn61] [BGC00]. Más tarde recibió aportaciones de otros, como Karnopp [Karn90] [Rose83] y Breedveld [Bree85]. Este último añadió principios basados en las leyes de la termodinámica. Los bond graphs modelan el contenido y la transferencia de energía de los sistemas, mediante la descripción de la distribución de la energía entre elementos físicos conectados. Los procesos físicos se representan como los nodos en un grafo dirigido [Cell91], [Thom89], y los enlaces (con forma de arpón) representan intercambio ideal de energía entre los nodos.

  Aunque en principio fueron usados principalmente para modelar sistemas mecánicos lineales, últimamente se han introducido nuevos conceptos que permiten el modelado de sistemas complejos por medio de Bond-graphs jerárquicos, siendo posible la no linealidad.

  En todos los sistemas físicos, la potencia se puede escribir como un producto de dos variables, una de tipo *"across"* (o variables de esfuerzo, tienen *"e"* como símbolo en los grafos), y otra de tipo *"through"* (variables de flujo, tienen una *"f"* como símbolo en los grafos)[Elmq78]. Siguiendo las leyes de Kirchoff, las variables de esfuerzo que llegan a un nodo deben tener el mismo valor, mientras que las variables de flujo suman cero. Estos enlaces, se conectan bien a elementos del sistema (que en esta terminología son elementos de un sólo puerto, tales como resistencias), o bien a empalmes. Hay dos tipos de empalmes:

  - de tipo 0, en el que todas las variables de esfuerzo son iguales y las de flujo suman cero (representa la ley de la corriente de Kirchoff).
  - de tipo 1, en la que todas las variables de flujo son iguales, mientras que las de esfuerzo suman cero (representa la ley del voltaje de Kirchoff).

  Además de las variables básicas (*e* y *f*), se usan a menudo la cantidad del momento generalizado, que se puede expresar como:

$$p = \int e \cdot d\tau$$

Fórmula II.1: momento generalizado.

además del desplazamiento generalizado, que se expresa como:

$$q = \int f \cdot d\tau$$

Fórmula II.2: desplazamiento generalizado.

Dependiendo del dominio del sistema físico, las variables e, f, p y q representarán magnitudes distintas, así, por ejemplo, en electricidad, estas variables representan el voltaje, la corriente, el flujo y la carga, respectivamente; en hidráulica, la presión, el flujo de volumen, el momento de presión y el volumen, etc.

Este paradigma ha sido adaptado a Modelica por J.F.Broenink [Broe97], [Broe99].

- **Grafos lineales [Chan91], [Well79]:** Es un grafo orientado similar a los bond graphs. Queda descrito por una estructura (grafo), una serie de leyes fenomenológicas aplicables a los elementos de la estructura, y una serie de parámetros aplicables a las leyes. Entre los sistemas que se basan en grafos lineales, se encuentran el *MSI* [Lore99] [MS197], y un prototipo para la simulación de sistemas mecatrónicos que se está desarrollando actualmente en la universidad de Carnegie-Mellon [Díaz99].

- **Grafos de dinámica de sistemas:** Surgieron en los sesenta, con el nombre de dinámica industrial, en el libro del mismo nombre de J. Forrester [Forr61]. Su propósito es explorar sistemas con retroalimentación en términos de estabilidad y respuesta a cambios externos [Pidd93]. Los grafos de dinámica de sistemas son diagramas en los que cada elemento del sistema tiene un símbolo (ver tabla II.1).

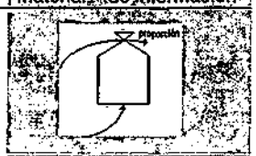
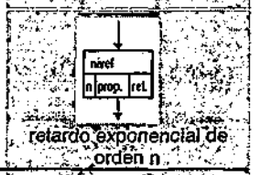
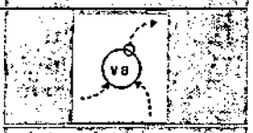
Nombre	Descripción	Símbolo
Nivel	Acumulaciones en el sistema	
Flujo	Movimiento de materiales e información en el sistema	
Funciones de decisión	La forma en la que se controlan los flujos.	
Retardos	Un caso especial de flujo. Ocurren cuando un elemento recibe información y la transmite un poco más tarde, o en menor proporción.	
Fuentes	El inicio de un flujo, si el flujo empieza desde fuera del sistema.	
Sumideros	El destino final de un flujo, si ese destino está fuera del modelo.	
Variables auxiliares	Se usan si conviene realizar alguna operación algebraica	
Parámetros	Constantes	

Tabla II.1: Símbolos usados por la dinámica de sistemas.

Un modelo de dinámica de sistemas está formado por un conjunto de ecuaciones en diferencias de primer orden, que reemplazan a las ecuaciones diferenciales. Hay varios tipos de ecuaciones:

- *ecuaciones de nivel*, que describen cómo cambian las variables de nivel con respecto al tiempo (cada variable de nivel tiene su ecuación de nivel).
- *ecuaciones de proporción*, que representan la política de administración de los cambios de proporción.
- *auxiliares*, se usan para dividir las ecuaciones de proporción, si éstas son muy complicadas.
- *suplementarias*, se introducen para definir variables que se necesitan para la salida, pero que no forman parte del modelo dinámico.
- *de valor inicial*, dan valores iniciales a las variables de nivel y a algunas proporciones.

Se pueden modelar varios tipos de retardos, por ejemplo:

- *retardos exponenciales de primer orden*, consisten en una variable de nivel, incrementada por una proporción de entrada, y mermada por una proporción de salida. Hay flujos de información que van de la variable de nivel a la función de decisión que controla la salida del flujo, también hay una retardo constante ejerciendo su influencia sobre dicha función.
- *retardos exponenciales de orden n*, siguiendo los mismos principios, se puede modelar un retardo de orden n conectando los flujos de n retardos de primer orden.

Entre los sistemas que se basan en diagramas de dinámica de sistemas están *STELLA* [Stel99], PowerSim [Powe99], ModelManager/ModelMaker [Mode99] y *AME* [Tayl99].

- **Ecuaciones matemáticas.** Pueden ser de dos tipos:

  - Causales o explícitos. En la parte izquierda de la igualdad se deja la variable de la que se quiere conocer el valor. Ejemplos de sistemas que usan este formalismo son *MIMIC* [Yoah69], *CSMP* [IBM72] [Spec76] y *OOCSMP* [Alfo97].

  - No causales. Es el sistema el que se encarga de "despejar" las variables que necesite. Ejemplos de lenguajes que usa este paradigma son Modelica [Mode99a] y *NMF* [Sahl89].

El ejemplo anterior, se representaría de las siguientes formas en alguno de estos formalismos (figuras II.6, II.7, II.8 y II.9):

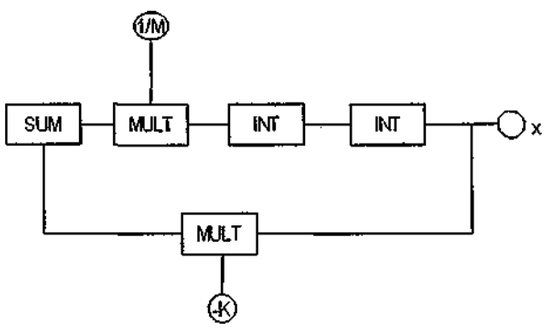


Figura II.6: Diagrama de bloques.

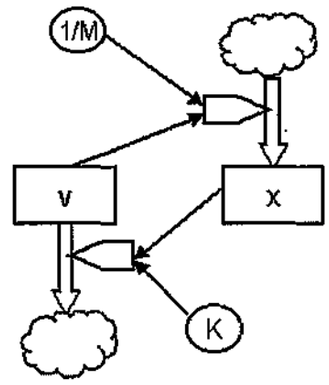


Figura II.7: Diagrama de dinámica de sistemas.

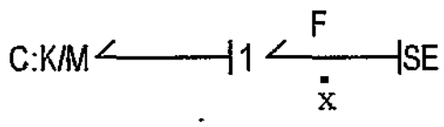


Figura II.8: Bond Graph.

$$x = x0 + \int_0^t v(t)dt$$

$$v = v0 + \int_0^t \frac{-Kx(t)}{M}dt$$

Figura II.9: Ecuaciones causales.

Se hace necesario un enfoque que cubra también las discontinuidades de los sistemas físicos, ya que la mayoría de ellos se comportan en parte de forma continua, y en parte de forma discreta. Fahrland [Fahr70] fue el primero en señalar este hecho. Ejemplos de sistemas físicos en los que ocurren discontinuidades, pueden ser:

- Un péndulo oscilante al que se le rompe la cuerda.
- En un ecosistema, cuando una especie emigra.

Dos tipos de eventos pueden producirse en la simulación mixta:

- Eventos temporales: son eventos que se planifican y ocurren en puntos específicos del tiempo.
- Eventos de estado: no se planifican, ocurren cuando el sistema llega a un estado particular.

Hay dos formas posibles de tratar este problema:

- Introducir bloques discontinuos.
- Tratar la discontinuidad como un evento (sistema híbrido) e inicializar otra vez el método de integración.

II.2 Simulación basada en la Web. Aprendizaje a distancia

La popularidad que está obteniendo Internet y el creciente número de ordenadores que están conectados a ella, está haciendo que muchas disciplinas estén reconsiderando sus técnicas tradicionales, para adaptarlas a las nuevas tecnologías [Page00]. Algunas de estas disciplinas son la simulación y la educación [Broo97].

Los entornos de aprendizaje interactivos son entornos para educación que facilitan el aprendizaje haciendo que los usuarios interaccionen con simulaciones de la materia a estudio [Bred98]. El uso de simulaciones en la enseñanza presenta aspectos positivos [deJo91], como por ejemplo en el caso de que el alumno no pueda acceder al el sistema real, ya sea debido a la distancia geográfica o de cualquier otro tipo. En este caso, la simulación de las características importantes del sistema mediante entornos de aprendizaje, y particularmente, si estos entornos contienen elementos multimedia [Scha95] puede ser un buen complemento a la simulación en el laboratorio.

La simulación interactiva y visual va a facilitar al alumno la comprensión de teorías, ya que permite el aprendizaje por exploración. Él mismo podrá diseñar sus experimentos, variando parámetros al comienzo o durante la simulación, e indudablemente hará más ameno el proceso de aprendizaje. Lo ideal sería poder incluir ciertos elementos que guíen al alumno en su exploración del problema, tales como sistemas de razonamiento cualitativo [Bred98], técnicas de *tutoring* [Alfo98a], etc.

En los sistemas de simulación, se puede manipular el tiempo, acelerando o ralentizando el comportamiento de los sistemas en estudio. Esto va a permitir a los alumnos acceder a nociones más globales de cómo el comportamiento de los sistemas evoluciona en el tiempo (por ejemplo, cambios en el clima, o procesos polutivos), o estudiar fenómenos complejos y de cambio rápido paso a paso en detalle (por ejemplo, reacciones químicas o fenómenos eléctricos).

La educación en la Universidad tiende a moverse desde un paradigma centrado en el profesor, a un paradigma centrado en el alumno [Maly98]. En este paradigma, el estudiante es más activo en el aprendizaje. La simulación basada en la web es un marco efectivo para un aprendizaje más activo. Tópicos como aprendizaje colaborativo [Mora00] o cursos adaptativos a cada perfil de usuario [Carr99] quedan fuera del alcance del presente trabajo.

El término *"simulación basada en la Web"* surgió a mediados de los 90, con tres artículos presentados en la 1996 Winter Conference. Dos de ellos presentaban simulaciones basadas en el lenguaje Java [Buss96], [Nair96]. El tercero [Fish96] se considera como el inicio de los posteriores desarrollos de los conceptos de simulación basada en la web.

Dentro de *"simulación basada en la Web"*, hay cabida para las siguientes áreas [Page98a]:

1) Simulación como un elemento hipermedia más, junto con texto, imágenes, vídeo, etc. La aplicación de este área a la educación a distancia es inmediata. Es posible la construcción de cursos educativos en diversas áreas (ingeniería, física, biología, etc.) que incluyan simulaciones interactivas [Schm99a], [Alfo99d].

2) Modelado distribuido. Este área incluye investigación en el desarrollo de herramientas y entornos que soporten el diseño colaborativo y distribuido de modelos de simulación a través de Internet [Cube98], [Fish98]. Los modelos residirían en los lugares donde pudieran ser más fácilmente catalogados, mantenidos, etc., o simplemente en el lugar donde han sido construidos.

3) Ejecución distribuida. Se incluyen actividades que usan Internet como infraestructura para el soporte de simulaciones distribuidas [Fox98], [Klei98], [Pag98b], [Alfo00a], [Alfo00b].

4) Acceso remoto a simulaciones. Dependiendo de cómo se acceda a los programas de simulación remotos, podemos distinguir:

   a) Servidor grueso (*thick server*) [Serb97]. Los programas de simulación se ejecutan en el servidor, y se programan en cualquier lenguaje accesible mediante *CGI* (Common Gateway Interface).

Este enfoque centraliza la ejecución de los modelos, pero incrementa el trafico de la red (si la simulación es interactiva), que puede generar problemas de rendimiento. Otro enfoque [Long99] sería el de usar la web para la configuración de simulaciones no interactivas de gran escala, el almacenamiento de sus resultados, así como su posterior análisis y visualización.

b) Cliente grueso (*thick client*) [Schm99a]. Las herramientas de simulación (interpretes, plug-ins, etc.) se tienen que pre-cargar o descargar en cada cliente, y se ejecutan allí. Este enfoque decrementa el tráfico de datos con el coste de requerimientos más severos en los clientes. Si se descargan programas ejecutables, hay peligro de incompatibilidad de la plataforma y de transmisión de virus.

c) Enfoque basado en navegador [Alfo99d]. Permiten conseguir los objetivos de 1). Los modelos se integran con las páginas *HTML* que componen los cursos, y se ejecutan como *applets* Java. El lenguaje Java tiene propiedades muy interesante en esta área, como la famosa frase *"write once, run everywhere"*, que proporciona a los programas independencia de la plataforma. Este enfoque implica que la carga inicial de las páginas será más lenta, porque se tienen que cargar los *applets*, pero la ejecución del modelo será más rápida.

5) Simulación de la web. Modelado y análisis de la web para la optimización y caracterización de su rendimiento.

De acuerdo con [Page99a] el uso de la web para la simulación, está haciendo surgir una serie de problemas y cuestiones interesante, entre ellas:

- Proliferación de objetos digitales. Probablemente en un futuro, los vendedores y fabricantes de objetos físicos emplearán modelos de simulación de estos objetos físicos, y estarán disponibles a través de la web [Fish98].

- Proliferación de estándares. La evolución de ciertos estándares es crucial para el desarrollo de los conceptos de interoperabilidad y modelado por composición, muy importantes en la simulación basada en la web (estos dos conceptos se explicarán más adelante). Estándares importantes para la simulación basada en la web son *UML* (Unified Modeling Language) [Erik98] [Lee97], *XML* (Extensible Markup Language) [XML97], *CORBA* (Common Object Request Broker Architecture) [Berg97] [Sieg96], y *HLA* (High Level Architecuture). Este último es un intento de estandarización para las simulaciones distribuidas que está llevando a cabo el ministerio de defensa de los Estados Unidos, primero llevado a cabo con el estándar *DIS* [DMSO99] (Distributed Interactive Simulation), y más tarde con el más avanzado *HLA* [DMSO99] [HLA99]. Ambos permiten que varios simuladores se conecten conjuntamente en un mundo virtual que comparten. De esta forma, es posible, por ejemplo que varios simuladores de vuelo, o de cualquier otro tipo, en distintos sitios interaccionen en el mismo paisaje. El tipo de simulaciones que se adaptan a *HLA* son principalmente de tipo militar, en el que existe interacción humana en la simulación (*human-in-the-loop simulation*).

- Construcción de modelos por composición. En una web poblada de objetos digitales, sería natural la búsqueda de objetos con los que construir nuestro modelo de simulación. La reutilización de componentes sería una consecuencia de este enfoque, tecnologías basadas en componentes, tales como *CORBA*, *COM/DCOM/OLE* [SEI00], *ActiveX* [Acti00] o *JavaBeans* [Srid97], jugarían un papel importante [Page99b]. El concepto de composición esta estrechamente ligado con el de interoperabilidad, que quiere decir que es posible la incorporación o baja de simuladores y objetos en tiempo de ejecución de la simulación. Este concepto se ha incorporado al estándar *HLA*, e implica unos interfaces comunes de acceso a los objetos de simulación.

- Resolución de problemas mediante el método de prueba y error. El uso de la web propicia un enfoque interactivo a la resolución de problemas, tanto en las ejecuciones de la simulación, en las que el usuario puede cambiar parámetros de forma descontrolada, tanto en el diseño de simulaciones mediante composición, en el que se puede ir probando distintas combinaciones de objetos que se han recogido de distintos lugares de la web.

- Uso de simulaciones por no expertos. Las simulaciones a través de la red estarán disponibles para un gran número de personas, que en su gran mayoría no serán expertos en simulación. Esto puede hacer necesario guías o agentes inteligentes para el uso de las simulaciones. Otro posible enfoque sería integrar sistemas de razonamiento cualitativo (*QR*) [Bred98] con los simuladores cuantitativos. Los sistemas *QR* pueden proporcionar explicaciones al usuario de lo que está sucediendo en la simulación en cada momento. Nuestro sistema no incorpora ninguno de estos elementos, sí bien en la sección IX.3 se muestra una aproximación (en la generación de código C++/Amulet) basada en presentar escenarios de problemas típicos al usuario. De esta forma, puede tener una cierta guía de los experimentos interesantes que es posible realizar. Un concepto parecido se presenta en la sección IV.3.3 con el diseño de simulaciones alternativas que se ejecutan en el mismo problema.

- Sistemas multi-lenguaje. La proliferación de objetos digitales en la red dará lugar a una heterogeneidad en los lenguajes en los que se han implementado estos objetos. La construcción de modelos por composición producirá entonces sistemas multi-lenguaje.

III. Sistemas anteriores

Este capítulo comienza describiendo *CSMP*, predecesor de nuestro lenguaje *OOCSMP* (sección 1). En las siguientes secciones se describen otros lenguajes y sistemas de simulación actuales. Si bien hay gran número de lenguajes y sistemas de simulación, se han querido destacar unos cuantos, debido, bien a su representatividad o bien a su similitud con nuestra idea. Estos son:

- Lenguaje Modelica [Mode99], sección 2.
- Lenguaje *NMF* [Sahl89], sección 3.
- *ASCEND* [ASCE91], sección 4.
- *AME* [Tayl99], sección 5.
- EcoSim [Ecos99], que también contiene un lenguaje de simulación, sección 6.
- Matlab 5.3 [Matl99] y SIMULINK 3.0 [Matl99], sección 7.
- *STELLA* [Stel99] y Systems Thinking [Rich94], sección 8.
- Dymola [Dymo99], sección 9.
- *MSI* [Lore99], sección 10.
- *OOPM* [Cube98], sección 11.

Continúa este capítulo, en la sección 12, con la descripción de otras ideas relacionadas que están surgiendo en la actualidad, tales como los Modelos de datos de Producto (*PDM*), el concepto de librería activa y de entorno para la resolución de problemas (*PSE*).

Finalmente se presenta en la sección 13 una justificación del lenguaje *OOCSMP*, una vez comparado con los distintos lenguajes y sistemas.

III.1 *CSMP*

III.1.1 Descripción general

CSMP (Continuous Systems Modelling Program) [IBM72], es un lenguaje de simulación continua, tipo *CSSL*, basado en ecuaciones matemáticas. Fue patrocinado por *IBM*, y tuvo su auge en los 70 y los 80.

Un programa *CSMP* se divide en secciones, que son:

- Sección de inicialización de parámetros (variables que no van a cambiar durante la simulación).
- Sección *INITIAL*, donde se dan valores iniciales a variables mediante expresiones. Se ejecuta sólo una vez, al inicio de la simulación.
- Sección *DYNAMIC*, que se ejecuta una vez por cada paso del intervalo elemental de integración.
- Sección de declaración de variables de control (paso elemental de tiempo, tiempo final, condiciones especiales de finalización, etc.), indicación de las variables que se imprimen, se dibujan, y elección del método de integración.

El elemento principal de un modelo *CSMP* son los bloques, que se pueden considerar elementos de un diagrama de bloques, o funciones matemáticas. Se incluyen en instrucciones de la sección *DYNAMIC* o *INITIAL* con sintaxis parecida a:

```
Y := INTGRL(IC,X)
```

que quiere decir que a la variable *Y* se le asigna la integral en el tiempo de la variable *X*, desde 0 a *TIME*. *IC* es la condición inicial de integración. Es decir, *Y* es el resultado de la ecuación diferencial:

```
X  := dY/dt
IC := Y(0)
```

CSMP es un lenguaje no secuencial, es decir que reordena las ecuaciones de las secciones *DYNAMIC* e *INITIAL* para que se ejecuten en el orden adecuado.

Los nombres de variables y constantes *CSMP* son de longitud máxima ocho, no pueden empezar por un dígito, ni coincidir con una palabra reservada. La siguiente tabla muestra las palabras reservadas de *CSMP*.

ADAMS	METHOD	RKS
DATA	MINdelta	RKSFX
DELTA	NOSORT	SIMP
DYNAMIC	PLOT	TIME
FINISH	PRdelta	TIMER
INITIAL	PRINT	TITLE
MAXdelta	RECT	TRAPZ

Tabla III.1: Palabras reservadas de *CSMP*

III.1.2 Elementos básicos

Los elementos básicos del lenguaje *CSMP* son:

- Funciones primitivas, son operaciones aritméticas que se indican mediante símbolos especiales, coinciden con las tradicionales de cualquier lenguaje de programación, y son:

Símbolo	Función
+	Suma.
-	Si es diádica, resta, si es monádica, negación.
*	Producto.
/	División.

Tabla III.2: Funciones primitivas de *CSMP*

- Funciones de bloque, representan operaciones matemáticas más complejas, se pueden combinar mediante expresiones primitivas, pero no pueden anidarse. Sus argumentos sólo pueden ser variables o constantes (no pueden ser expresiones). Las siguientes tablas muestran los bloques disponibles en *CSMP*.

Descripción	Sintaxis
Seno	Y := SIN (X)
Coseno	Y := COS (X)
Tangente	Y:= TAN (X)
Arcoseno	Y := ARSIN (X)
Arcocoseno	Y := ARCOS (X)
Arcotangente	Y := ATAN (X)
Seno hiperbólico	Y := SH (X)
Coseno hiperbólico	Y := CH (X)
Tangente hiperbólica	Y := TANH (X)
Arcoseno hiperbólico	Y := ARCSH (X)
Arcocoseno hiperbólico	Y := ARCCH (X)
Arcotangente hiperbólica	Y := ARCTH (X)
Exponencial	Y := EXP (X)
Logaritmo neperiano	Y := LOG (X)
Logaritmo decimal	Y := LOG10 (X)
Raíz cuadrada	Y := SQRT (X)
Función gamma	Y := GAMMA (X)
Valor absoluto	Y := ABS (X)
Valor mayor	Y := MAX (X1, X2, ..., Xn)
Valor más pequeño	Y := MIN (X1, X2, ..., Xn)

Tabla III.3: Funciones aritméticas.

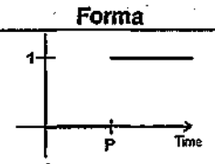
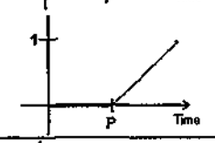
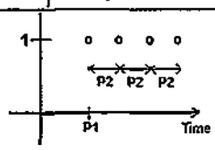
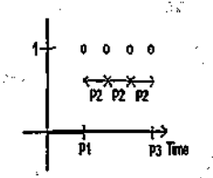
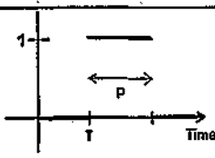
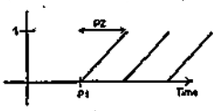
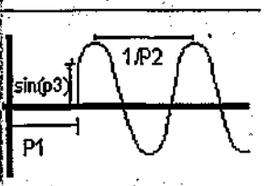
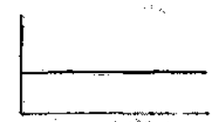
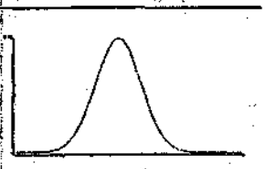
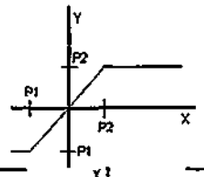
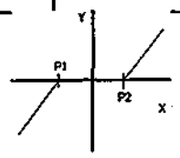
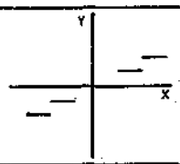
Nombre	Sintaxis	Descripción	Forma
Función Escalón	Y:=STEP(P)	if (TIME<P) then Y:=0 else if (TIME>=P) then Y:=1	
Función Rampa	Y:=RAMP(P)	if (TIME<P) then Y:=0 else if (TIME>=P) then Y:=TIME-P	
Generador de Impulsos	Y := IMPULS (P1,P2)	if (TIME<P1) then Y:=0 else if (TIME-P1)=K*P2 then Y:=1 else if (TIME-P1)!=K*P2 then Y:=0 para K=0,1,2...	
Función de muestreo	Y := SAMPLE (P1, P2, P3)	if (TIME<P1) then Y:=0 else if (TIME>=P1+K*P3)<=P2 then Y:=1 else Y:=0 para K=0,1,2... P1 = tiempo inicial de muestreo. P2 = tiempo final de muestreo. P3 = intervalo de tiempo entre muestras.	
Generador de pulsos	Y := PULSE (P, X)	if T<=TIME<(T+P) or X>0 then Y:=1 else Y:=0 T es el instante en el que X se hace mayor que cero, cuando Y vale 0.	
Generador de dientes de sierra	Y := SAWTH (P1, P2)	if (TIME<=P1) then Y:=0 else Y:=(TIME-P1)/P2	
Onda sinusoidal	Y := SINE(P1,P2,P3)	if (TIME<=P1) then Y:=0 else if (TIME>=P1) then Y:=SIN(P3+P2*(TIME-P1)) P1 = retardo. P2 = velocidad angular en radianes por unidad de tiempo. P3 = desfase en radianes.	
Generador de ruido aleatorio (distribución uniforme)	Y:=RNDGEN(P)	Y:=0.0001*RANDOM(10000) Y ∈ [0,1] 4 cifras significativas P se ignora.	
Generador de ruido aleatorio (distribución Gaussiana)	Y:=GAUSS(P1, P2)	$Y = \frac{1}{P2\sqrt{2\Pi}} e^{-\frac{1}{2}\frac{(X-P1)^2}{P2^2}}$ P1 = media. P2 = desviación estándar.	

Tabla III.4: Generadores de funciones de onda.

Nombre	Sintaxis	Descripción	Forma

Limitador	Y := LIMIT(P1,P2,X)	if (X<P1) Y:=P1 else
		if (X>P2) Y:=P2 else
		Y:=X
			

Zona muerta	Y := DEADSP(P1,P2,X)	if (X<P1) Y:=X-P1 else
		if (X>P2) Y:=X-P2 else
		Y:=0
			

Cuantificador	Y := QNTZR(P,X)	if (((K-0.5)*P<X) AND (X<=(K+0.5)*P)) Y:=K*P Para K=0,+1,-1,2,...	

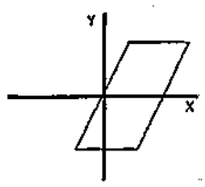
Ciclo de histéresis	Y := HSTRSS(IC,P1,P2,X)	if (((X(TIME)-X(TIME-delta))>0)
		AND (Y(TIME-delta)<=X-P2))
		Y:=X-P2
		else
		if (((X(TIME)-X(TIME-delta))<0) AND
		(Y(TIME-delta)>=X-P1))
		Y:=X-P1
		else Y:=Y(TIME-delta)
		
		IC = Condición inicial
		P1,P2 = Intersecciones con el eje X.
			

Tabla III.5: Modificadores de funciones de onda.

Nombre	Sintaxis	Descripción
Interpolación Lineal	Y:=AFGEN(F,X) (F es un vector)	Y := F(X)
Interpolación cuadrática	Y:=NLFGEN(F,X) (F es un vector)	Y:=F(X)
Interpolación de grado variable	Y:=FUNGEN(F,P,X) (F es un vector, P sólo puede valer 1 ó 2)	Y:=F(X)
Pendiente de una curva	Y:=SLOPE(F,P,X) (F es un vector, P sólo puede valer 1 ó 2)	Y:=DF(X)/DX

Tabla III.6: Generadores de funciones arbitrarias.

Nombre	Sintaxis
No lógico	Y := NOT(X)
Y lógico	Y:=AND(X1, X2, ..., Xn)
O inclusivo	Y := IOR(X1,X2, ..., Xn)
O exclusivo	Y := EOR(X1, X2)
No Y	Y:=NAND(X1,X2, ..., Xn)
NO O inclusivo	Y := NOR(X1,X2, ..., Xn)
Flip Flop RST	Y := RST (X1,X2,X3)
Flip Flop JK	Y:=FFJK(X1,X2)

Tabla III.7: Bloques lógicos.

Nombre	Sintaxis	Descripción
Conmutador de entrada	Y:=INSW(X1,X2,X3)	if (X1<0) then Y:=X2 else Y:=X3
Conmutador de función	Y:=FCNSW(X1,X2,X3,X4)	if (X1<0) then Y:=X2 else if (X1==0) then Y:=X3 else Y:=X4
Conmutador de salida	Y:=OUTSW(X1,X2)	if (X1<0) then Y:=X2,0 else if (X1>=0) then Y:=0,X2 (e.d. el resultado de este bloque es un vector de dos elementos)
Retardo de orden 0	Y:=ZHOLD(X1,X2)	if (X1>0) then Y:=X2 else if (X1<=0) then Y:=Y(T-delta)

Tabla III.8: Conmutadores.

Nombre	Sintaxis	Descripción
Integrador	Y:=INTGRL(IC,X)	$Y := IC + \int_0^{TIME} X \bullet dt$
Derivada	Y:=DERIV(IC, X)	Y:=(X(T)-X(T-delta))/delta
Retardo	Y:=DELAY(N,P, X)	if (TIME<P) then Y:=0 else if (TIME>=P) then Y:=X(TIME-P) N es una constante numérica mayor que uno, e indica el número de puntos muestreados en el intervalo P (el retardo)
Función Implícita	Y:=IMPL(IC,P,FY)	Y:=FY(Y) además |Y-FY(Y)|<=|P*Y| IC = Primera aproximación P = Cota del error FY = Función de Y
Comparador	Y:=COMPAR(X1,X2)	if (X1<X2) then Y:=0 else if (X1>=X2) then Y:=1
Equivalencia	Y:=EQUIV(X1,X2)	if ((X1<=0) AND (X2<=0)) then Y:=1 else if ((X1>0) AND (X2>0)) then Y:=1 else Y:=0

Tabla III.9: Otros bloques.

■ III.1.3 Instrucciones de estructura

Un programa *CSMP* se divide en los segmentos comentados en el apartado III.1. Un segmento está compuesto de asignaciones de expresiones a variables. Las asignaciones dentro de un segmento se reordenan, para ser ejecutadas en el orden correcto, excepto si se indica lo contrario con el mandato *NOSORT*.

■ III.1.4 Instrucciones de declaración de datos

Las instrucciones de declaración de datos comienzan por la palabra reservada DATA, seguido de una lista de asignaciones de valores constantes a parámetros, separadas por comas. Estos parámetros no pueden cambiar su valor durante la ejecución de la simulación.

III.1.5 Instrucciones de control

Las instrucciones de control se usan para asignar valores a las variables que se ocupan del manejo de la simulación, para definir el método de integración, establecer condiciones de finalización, o bien para configurar la salida.

Para asignar valores a las variables de control, se usa la palabra reservada *TIMER*, seguido de una lista de asignaciones de valores a alguna de las siguientes variables:

- *FINTIM*, es el valor final de la variable *TIME* en una simulación.
- *delta*, es el intervalo elemental de tiempo.
- *PRdelta*, intervalo de impresión de variables.
- *PLdelta*, intervalo para el dibujo de variables.
- *MAXdelta*, fija un valor máximo para delta, cuando el sistema varía delta automáticamente (sólo con el método *RKS*).
- *MINdelta*, fija un valor mínimo para delta, cuando el sistema varía delta automáticamente (sólo con el método *RKS*).

El método de integración se selecciona mediante la palabra reservada *METHOD*, seguido de:

Nombre	Método
RECT	Rectangular
ADAMS	Adams de segundo orden, intervalo fijo
TRAPZ	Trapezoidal con intervalo fijo
SIMP	Regla de Simpson con intervalo fijo
RKSFX	Runge-Kutta de cuarto orden con intervalo fijo
RKS	Runge-Kutta de cuarto orden con intervalo variable

Tabla III.10: Métodos de integración.

Las condiciones de finalización que no son dependientes explícitamente de *TIME*, se declaran mediante la instrucción *FINISH* seguido de *<variable-1> = <variable-2>* o de *<variable-1> = <constante>*. La ejecución de la simulación termina cuando *<variable-1>-<variable-2>* (o bien *<variable-1>-<constante>*) cambia de signo.

Las variables que se han de imprimir se declaran a continuación del mandato *PRINT*, separadas por comas. Para dibujar gráficamente valores de variables, se usa la instrucción *PLOT*, seguido del alto y ancho del gráfico y a continuación, las variables. La última variable se considera la variable de abscisa.

Se puede poner un título a los gráficos y a la salida impresa mediante el mandato *TITLE*, seguido del título.

Los comentarios empiezan por '*'.

III.1.6 Bucles cerrados

Si aparece un bucle cerrado en una sección, en éste debe aparecer al menos un bloque con memoria (*INTGRL* o *DELAY*).

III.1.7 Ejemplo

Supongamos que queremos estudiar el comportamiento de un muelle. Para modelar este sistema, vamos a hacer una serie de simplificaciones: por ejemplo, no se va a considerar la fricción, y vamos a reducir el problema a una dimensión. Las magnitudes que vamos a estudiar son la posición del extremo, la velocidad y la energía total del sistema.

Partiendo de la ley de Hook, tenemos que:

$$F = -K \cdot x$$

Ecuación III.1: Ley de Hook

Por la ley del movimiento de Newton, y por III.1:

$$F = M \cdot a$$

$$a = \frac{-Kx}{M}$$

Ecuación III.2: Ley del movimiento de Newton

Además:

$$a = \frac{dv}{dt}, v = \frac{dx}{dt}$$

Ecuación III.3: Aceleración y velocidad.

La Energía total de un sistema es:

$$\mathrm{Et} = \mathrm{Ec} + \mathrm{Ep}$$

$$\mathrm{Ec} = \frac{1}{2}\mathrm{mv}^2$$

$$\mathrm{Ep} = \frac{1}{2}\mathrm{hF}$$

Ecuación III.4: Energía total de un sistema.

Poniendo las ecuaciones III.3 en forma causal, obtenemos:

$$v = v_0 + \int_0^{TIME} \frac{-Kx}{M} dt$$

$$x = x_0 + \int_0^{TIME} v dt$$

Ecuación III.5: Forma causal de las ecuaciones III.3

De esta forma, ya podemos programar nuestro modelo *CSMP*, que quedaría de la siguiente forma:

```
TITLE HARMONIC 1D WITHOUT FRICTION
DATA K:=4.0, V0:= 1, M:= 1
DYNAMIC
    F:=-K*X
    F1:=F/M
    V:=INTGRL(V0,F1)
    X:=INTGRL(X0,V)
    EC:=M*V*V/2
    EP:=K*X*X/2
    ET:=EC+EP

TIMER delta:=0.1, FINTIM:=10, PLdelta:=0.1, PRdelta:=1
PLOT  X,V,ET,TIME
PRINT X,V,ET
METHOD RECT
```

Listado III.1: Modelo *CSMP* de un muelle sin fricción.

La siguiente figura muestra el resultado de la ejecución de la simulación, una vez compilado el modelo con *C-OOL* (el compilador que se ha construido para *OOCSMP*, pero que es compatible con los modelos *CSMP*).

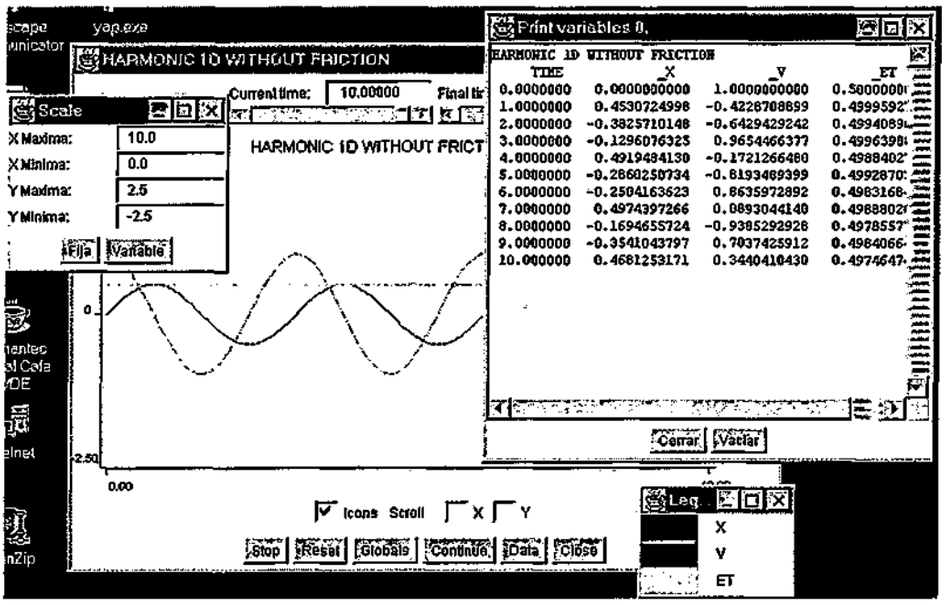


Figura III.1: Resultado de la simulación del modelo del muelle.

III.2 Modelica

El lenguaje Modelica surgió en 1997 como un intento de estandarización de los lenguajes de simulación. Es un lenguaje de simulación digital continua orientado a objetos, que usa el paradigma de ecuaciones no causales. Hasta hace unos meses no había una implementación de Modelica en el mercado, actualmente la única implementación es la de la empresa sueca Dymola AB [Dymo99], que lo incorpora en la versión 4.0 de su producto Dynasim.

Hasta ahora han surgido cuatro versiones de este lenguaje:

- Modelica v1.0: Apareció en Septiembre de 1997.
- Modelica v1.1: Apareció en Diciembre de 1998.
- Modelica v1.2: de Junio de 1999.

Como trabajo futuro de esta tesis, está prevista la generación de código Modelica a partir de código *OOCSMP*, si bien Modelica presenta algunas limitaciones con respecto a *OOCSMP*:

- En principio, Modelica no tiene prevista la simulación distribuida.
- Carece de primitivas para el manejo de elementos multimedia.
- El control de las salidas gráficas se deja al sistema que interpreta el lenguaje Modelica.
- Los bloques del lenguaje no están 'sobrecargados', sólo se pueden usar con escalares.
- No es posible aplicar un método de un objeto a una colección de objetos, como en *OOCSMP*. Con esta limitación sería complicada la implementación en Modelica de la simulación del sistema solar, que se presenta en el curso de gravitación, del apartado X.
- Tampoco son posibles expresiones del tipo `<escalar> += <vector>` que sí son posibles en *OOCSMP*, y que generan un bucle implícito.
- No soporta por el momento ecuaciones en derivadas parciales. Si bien tienen previsto incorporarlas en el futuro.

Sin embargo, Modelica tiene algunas características que lo hacen interesante:

- Sus ecuaciones son no causales.
- Los parámetros de las funciones y procedimientos se pueden definir como de entrada o de salida.
- Se pueden declarar tipos, soporta unidades de medida, si bien no hace un chequeo de que en una ecuación, las unidades sean correctas. También se pueden indicar los valores máximos y mínimos que pueden tomar las variables.
- Implementa directivas de compilación.
- Implementa la función *connect* y el tipo *connector*. Con estas dos construcciones es fácil traducir un modelo escrito mediante "*bond graphs*" al lenguaje Modelica.
- Se han construido bibliotecas para la simulación en diversas áreas de aplicación.
- Buen soporte para la documentación de los modelos.
- Soporte de discontinuidades.

Cuando se inició el trabajo de la presente tesis no existía el lenguaje Modelica. Cuando surgió el proyecto de Modelica, nuestro grupo fue miembro colaborador en el diseño del lenguaje, en la actualidad nuestro grupo pertenece al proyecto como 'miembro interesado'.

III.3 *NMF*

El *NMF* (Neutral Model Format) es un lenguaje de simulación, propuesto en 1989 en la conferencia Building Simulation '89 en Vancouver [Sahl89] como un medio de documentación e intercambio de modelos. Fue un intento de estandarización anterior a Modelica. Este lenguaje está controlado por el *ASHRAE* (American Society for Heating, Refrigerating and Air-Conditioning Engineers) comité técnico *TC* 4.7. Algunas de sus características son:

- Buen soporte para la documentación de modelos.
- Soporte para llamadas a código en Fortran o C.
- Dimensiones dinámicas de vectores.
- Ecuaciones en formato no causal.
- Posibilidad de declarar si las variables son de entrada o de salida.
- Soporte de construcciones similares a los *connectors* de Modelica.
- Se han propuesto extensiones a *NMF* para el soporte de herencia de modelos y de modelos híbridos continuos/discretos.

Algunos de los sistemas que usan *NMF* son: *IDA* [Bris99] [Sahl99], *SPARK*[Nata94], *ESACAP*, *TRNSYS*, *HVACSIM+* y *MS1* [Lore94], [Lore99]. Las limitaciones de *NMF* respecto a *OOCSMP* son básicamente las mismas que para Modelica.

III.4 *ASCEND*

El sistema de simulación *ASCEND* (Advanced System for Computation ENgineering Design), se desarrolló en la universidad de Carnegie Mellon, PA, USA, como un entorno para el desarrollo rápido de modelos complejos, basados en un número grande de ecuaciones algebraicas no lineales. Pone gran énfasis en el chequeo de unidades de medida.

Las características principales de *ASCEND* son:

- Soporte de tipos de variables y constantes "*ATOMS*" similares a las de Modelica (nombre de unidad, límite inferior y superior), pero con chequeo de los tipos.
- Soporte de ecuaciones no causales.
- Configuración del formato de la salida gráfica complejo (se trata como un modelo más), aunque flexible. Posibilidad de heredar y refinar modelos de dibujo.
- Chequeo de qué librerías necesita un fichero, y de qué librerías provee (*REQUIRE* y *PROVIDE*).
- Librerías para uso en distintas áreas de aplicación.
- Posibilidad de configurar la estructura del modelo condicionalmente, con la construcción *WHEN*.
- Buen soporte para la documentación de los modelos.
- Soporta discontinuidades

Las desventajas respecto al lenguaje *OOCSMP* serían las mismas que las desventajas de Modelica. Además, respecto al entorno de desarrollo *OOCSMP*, *C-OOL*, el sistema *ASCEND* tiene algunas limitaciones, tales como:

- Lenguaje y sistema están integrados, no se genera código objeto, lo que no permite a los usuarios usar una simulación si no tienen el sistema instalado en su ordenador.

III.5 *AME*

El propósito del sistema de simulación *AME* (Agroforestry Modelling Environment) es facilitar el proceso de construcción de modelos agroforestales. Utiliza como entrada un grafo de dinámica de sistemas. Las principales características de *AME* son:

- Genera código C o Tcl para la simulación.
- Genera páginas *HTML* con la documentación del modelo.
- Maneja discontinuidades mediante construcciones tipo *IF...THEN...ELSE*.

Si bien, tiene algunas limitaciones (a parte de las ya mencionadas para Modelica), como estar orientado a modelado de sistemas agroforestales.

III.6 *ECOSIM*

Es una herramienta para la simulación continua de sistemas, implementado en C.

- Tiene como entrada gráfica un diagrama de bloques, o bien un lenguaje, *EL*.
- Desde el lenguaje *EL* se puede llamar a rutinas C o *FORTRAN*.
- Se pueden modificar y ver variables durante la simulación.
- Las ecuaciones se introducen en forma no causal.
- Maneja discontinuidades.
- Interfaz que permite la depuración de simulaciones.
- Soporte de puertos (*PORTS*) muy parecidos a los tipos *connector* de Modelica.
- Soporte de Componentes, con filosofía parecida a la de Orientación a Objetos.

Las desventajas de *ECOSIM* son las mismas que las de Modelica y *ASCEND*.

III.7 Matlab 5.3 y Simulink 3.0

MATLAB es un entorno integrado para el cómputo técnico, que combina cálculo numérico y simbólico. Tiene un lenguaje de programación propio. Simulink usa Matlab, y es una herramienta más orientada a la simulación. La entrada de Simulink son diagramas de bloques. Algunas de las características de Matlab y Simulink son:

Matlab:
- Análisis de Fourier y estadístico.
- Manipulación de matrices dispersas.
- Gráficos avanzados.
- Existen muchos paquetes en el mercado que aumentan las capacidades de Matlab, tales como:
  - Paquete para la solución (incluyendo preproceso y post-procesp) de PDEs en dos dimensiones y el tiempo, mediante Elementos Finitos.
  - Optimización y control.
  - Redes Neuronales.
  - Lógica difusa, etc.

Simulink:
- Soporta simulación híbrida (continua y discreta).
- Posibilidad de llamada a código escrito en C.
- Modelado jerárquico.
- Soporte de números complejos.

Aunque Matlab/Simulink es muy completo, tiene alguna limitación con respecto a nuestros objetivos iniciales y son:

- No es posible generar código Java y páginas *HTML* para la generación de cursos.
- No soporta la ejecución distribuida de experimentos.
- No soporta elementos multimedia.
- No es orientado a objetos.
- Matlab y Simulink están integrados, no se genera código objeto, lo que no permite a los usuarios de la simulación usar un modelo si no tienen el sistema instalado en su PC.

III.8 *STELLA* 5.1.1 y Systems Thinking

El sistema de simulación *STELLA* (Structural Thinking, Experiential Learning Laboratory with Animation) está basado en diagramas de dinámica de sistemas. *STELLA* forma parte de la metodología Systems Thinking. Esta metodología fue desarrollada por Barry Richmond [Rich94], trata de hacer fiables las inferencias acerca del comportamiento de un sistema, desarrollando progresivamente un entendimiento más profundo de este sistema. Algunas de sus limitaciones son:

- Sólo incluye un formato de salida gráfica.
- No tiene tratamiento de eventos discretos.

III.9 Dymola

El sistema Dymola (DYnamic MOdelling LAnguage) contiene un lenguaje de simulación de propósito general y un entorno para la simulación de sistemas grandes. Ha sido desarrollado por la compañía sueca Dynasim AB. Algunas de sus características son:

- Soporte del lenguaje Modelica. Si bien se puede modelar también el problema gráficamente.
- Interfaz con Matlab/Simulink.
- Documentación *HTML* de los modelos.
- Solución simbólica de las ecuaciones cuando es posible.
- Posibilidad de generar C o Fortran.
- Soporte de eventos discretos.

Las desventajas respecto a *OOCSMP* serían las mismas que las de Modelica.

III.10 *MS1*

MS1 (Modelling System 1) es un entorno de modelado y simulación que soporta ecuaciones diferenciales ordinarias. Los modelos se pueden describir en *MS1* mediante *NMF*, diagramas de bloques, bond graphs, o grafos lineales.

Las limitaciones de *MS1* respecto a *OOCSMP* serían las mismas que las de *NMF*.

III.11 *OOPM*

OOPM (Object-Oriented Physical Multimodeling) [Cube98], surgió en 1998 en la universidad de Florida. Es una herramienta gráfica para el diseño de modelos de simulación que provee un repositorio de modelos distribuido, que facilita el reuso de modelos. Es posible generar código C++ con los modelos, además se puede ejecutar el código C++ con una interfaz que usa realidad virtual (*VRML*).

La forma de expresar el comportamiento de un sistema es mediante multimodelos [Fish92], [Fish93], que consiste en la conexión de modelos, que pueden estar expresados en distintos formalismos, como máquinas de estado finito, diagramas de bloques, ecuaciones diferenciales-algebraicas, modelos basados en reglas, grafos de dinámica de sistemas o código nativo C++.

Otras características de *OOPM* son:

- Cada clase *OOPM* puede tener asociados objetos gráficos codificados en *VRML*.

- El repositorio de modelos es distribuido, lo que permite la reutilización de modelos a través de la Web.
- Planean la migración de todo el sistema a tecnologías basadas puramente en navegación Web: plug-ins en los clientes, y el repositorio accesible mediante *CGI*.

El enfoque de esta herramienta es similar al presente trabajo, si bien *OOPM* presenta algunas limitaciones en cuanto a nuestros objetivos:

- Nuestro enfoque ha sido generar simulaciones ejecutables en la Web, mediante un compilador tradicional, justo lo contrario que ahora hace *OOPM*: la construcción de modelos es distribuida, pero la ejecución es en código C++, no ejecutable a través de la Web. Además se necesita tener instalado *OOPM* y las librerías Tcl/Tk para ejecutar los modelos.
- Al generar código C++, no permite la inclusión de estos programas en cursos para la Web.
- Los diversos formalismos que ofrece no son capaces de resolver ecuaciones en derivadas parciales.
- No permite la generación de código distribuido.
- No contempla la inclusión de elementos multimedia en las simulaciones.

III.12 Otros

Una herramienta de propósito similar a la expuesta en el presente trabajo, es el *VCLab* (Virtual Control Laboratory) [Schi99a] [Schi99b]. El objetivo de esta herramienta es presentar cursos educativos de ingeniería al alumno, a través de Internet. Para ello, integra *MATLAB/SIMULINK* como plug-ins de un navegador web, además de modelos de realidad virtual y multimedia. Aunque la filosofía es parecida a la de este trabajo, el esquema que presentan tiene ciertas limitaciones, ya que *MATLAB* no incluye primitivas de control de multimedia, debiéndose programar "ad-hoc" manualmente el enlace de la simulación con el modelo de realidad virtual y demás elementos multimedia. Además la herramienta no presta ayuda a la construcción del curso si no sólo a la construcción del modelo.

Otra herramienta de similares características a *VCLab* es *VR-Class* [Garc99]. El objetivo de esta herramienta es crear un entorno virtual que sirva a los alumnos como punto de encuentro, de forma que sea posible su colaboración y participación. La herramienta soporta (siempre dentro de un entorno virtual *VRML*) presentaciones 2-D y 3-D, gráficos en 3-D, y además planean construir una herramienta para la simulación en 3-D.

Los entornos para la resolución de problemas [Aker97] (*PSEs*) son sistemas que proveen facilidades para resolver una determinada clase de problemas. Muchos de estos *PSEs* están orientados a la solución de *PDEs*. Estas facilidades son típicamente métodos de resolución, selección automática o semi-automática de estos métodos y posibilidad de fácil incorporación de nuevos métodos. Además la descripción del problema se realiza en un lenguaje de muy alto nivel. A partir de estas descripciones, los sistemas *PSEs* generan programas de ordenador lo más optimizados posible que resuelven el problema. En general, los sistemas *PSEs*, no dejan de ser lenguajes de alto nivel orientados a las matemáticas (que no de simulación). En este sentido, *OOCSMP* tendría la ventaja de no estar orientado solamente a la resolución de *PDEs*, sino que es un lenguaje mucho más general. Sin embargo la sintaxis de *OOCSMP* es algo más complicada que la sintaxis típica de un sistema *PSE*, aunque básicamente los pasos que hay que dar para resolver una *PDE* con *OOCSMP* son los mismos que los que habría que dar con un *PSE*.

El término 'librerías activas' [Veld98] engloba librerías que toman un papel activo en la generación de código, la optimización e interaccionan con las herramientas de programación. A continuación se listan unos cuantos tópicos cubiertos por este tipo de librerías:

- Optimización de operaciones con operadores sobrecargados que involucran a vectores o matrices. Esta característica la implementa Blitz++ [Blit98], en parte también ha sido implementada en *OOCSMP* (ver sección IV.2).
- Desenrolle de bucles si estos son pequeños. Esta característica también la implementa Blitz++ [Blit98]. En este trabajo no se ha considerado, ya que esto suelen hacerlo los compiladores de C++ o Java.

- Generación de código paralelo para operaciones con matrices, esta característica la implementa la librería *POOMA* [Karm98]. En *OOCSMP*, se ha optado por otro tipo de paralelismo, de más alto nivel (ver apartado VII)
- Optimización del uso de matrices dispersas. Librerías que las tratan son *MTL* [Siek98] y *GMCL* [GMCL00]. Este aspecto no ha sido cubierto por *OOCSMP*.
- Lenguajes multinivel, que permiten evaluar expresiones bien en tiempo de compilación, bien en tiempo de ejecución [Stic97]. Aunque en cierto modo *OOCSMP* es un lenguaje multinivel, no tiene construcciones para indicar el momento de la evaluación, desde el nivel *SODA* se pueden evaluar expresiones que involucren valores iniciales de variables y constantes del nivel *OOCSMP*.
- Generación de código en tiempo de ejecución (*RTGC*), hay sistemas, como 'C [Engl96] y Fabius [Leon94] que son capaces de generar código optimizado, usando información que no está disponible hasta el momento de la ejecución. Esta característica tampoco ha sido implementada en *OOCSMP*.

Si bien quedan fuera del alcance del presente trabajo, también se han de mencionar:

- Las herramientas que implementan correspondencias entre la geometría del objeto, otras propiedades de interés, como su precio, tipo de material, etc., y las ecuaciones que gobiernan su comportamiento.

  Normalmente estas herramientas tratan de que todas la información sobre cada objeto de la simulación (por ejemplo, una determinada pieza), esté normalizada y resida en una base de datos, con acceso normalizado para todos los miembros de un equipo de trabajo interdisciplinar. Esta base de datos contiene toda la información necesaria para realizar simulaciones de muy distinta índole con las propiedades de cada objeto.

  Actualmente, se encuentra en desarrollo el Estándar para el intercambio de Modelos de Datos de Producto (Product Data Model, *PDM*), que se conoce como *STEP* (STandard for the Exchange of Product model data, *ISO* TC 184, 1993). El propósito de este esfuerzo es que diferentes herramientas de diverso uso puedan acceder a esta base de datos, manteniéndose en todo caso la coherencia e integridad de los modelos.

  Si bien esta clase de base de datos única, es hoy por hoy utópica, existen algunas herramientas y prototipos que implementan parcialmente algunas de estas capacidades, como por ejemplo:

  - *IDA* [Bris99], [Sahl99], esta herramienta de la empresa Bris Data hace una correspondencia de la geometría de los objetos a sus propiedades y ecuaciones. Es decir, mediante una descripción de la geometría del sistema a simular, y de cada uno de sus componentes, es posible obtener de una manera semiautomática las ecuaciones que lo gobiernan, en formato *NMF*. Este sistema no implementa una base de datos tipo *PDM*.

III.13 Justificación

En general, hemos observado que ninguno de los lenguajes o sistemas se adecuaba del todo a nuestras necesidades, que básicamente eran:

- Necesidad de generar código Java y páginas *HTML* para la creación de cursos educativos que contuvieran problemas de simulación. Además con la posibilidad de especificar la apariencia de las páginas *HTML* dentro del mismo entorno de simulación. Nuestro trabajo está enfocado hacia la construcción de una herramienta de autor para la construcción de cursos basados en simulación para la Web.
- Enriquecer la simulación con elementos multimedia.
- Combinar la orientación a objetos con la simulación, en particular con la resolución de ecuaciones en derivadas parciales.
- Interfaz de usuario flexible, con múltiples formatos de salida, que nos permita explorar el problema, fácilmente cambiable. De algún modo esta interfaz debería permitir pequeños cambios en el modelo que permitan un diseño flexible de experimentos.

- Flexibilidad en la sintaxis del lenguaje.
- Mantenibilidad del modelo de simulación, mediante la generación automática de documentación.

Para cubrir estas necesidades, surgió la idea de diseñar el lenguaje *OOCSMP*, al que posteriormente se le han ido añadiendo otras características, tales como la posibilidad de generar simulaciones distribuidas, o posibilidad de manejar eventos discretos.

IV. Extensiones básicas y orientadas a objetos

En este capítulo se van a tratar las extensiones orientadas a objetos que se han añadido al lenguaje *CSMP*. Estas y otras extensiones han dado lugar al lenguaje *OOCSMP*. Parte de estas extensiones se han tratado en los artículos [Alfo97], [Alfo99a], [Alfo99b]. También se presentan otras extensiones, tales como la ampliación de los tipos básicos para soportar un álgebra de vectores y matrices, la posibilidad de pasar parámetros externos a la simulación, etc.

Comienza el capítulo con la ampliación de los tipos básicos y de los bloques del lenguaje *CSMP*. Al final de esta sección, se presenta un ejemplo que clarifica las extensiones expuestas y muestra las capacidades de *OOCSMP* de realizar también simulación discreta: el autómata celular del juego de la vida.

La siguiente sección presenta las extensiones orientadas a objetos que hemos añadido al *CSMP*, comenzando con una introducción a la orientación a objetos, para seguir con los elementos básicos de la programación a objetos que hemos incorporado a *CSMP*:

- Clases *OOCSMP* (IV.8.2)
- Objetos *OOCSMP* (IV.8.3)
- Colecciones de objetos (IV.8.4).
- Métodos (IV.8.5)

La sección IV.9 trata otras extensiones añadidas al *CSMP*, tales como la inclusión de ficheros, parámetros externos al modelo, etc.

IV.1 Vectores y matrices

Se han ampliado los tipos básicos del lenguaje *CSMP*, para implementar un álgebra de vectores y matrices. Cuando se declaran vectores o matrices, se ha de indicar la dimensión, que ha de ser una constante numérica o una variable, y se tienen que declarar dentro de la instrucción *DATA*. Los índices, tanto de los vectores, como de las matrices, empiezan en 0. El siguiente ejemplo muestra la declaración de un vector y una matriz.

```
DATA vec[10], matr[12;20]
```

Ejemplo IV.1: Declaración de vectores y matrices.

Se puede asignar valor inicial a los vectores y matrices dentro de una instrucción *DATA*, después de su declaración. El valor puede ser asignado en forma de una lista de constantes, o bien como una expresión, que puede depender de las variables *ROW* y *COL*, que generan un bucle implícito sobre filas y columnas. El siguiente ejemplo muestra una asignación de valores iniciales a vectores y matrices:

```
DATA v[5], v[]:= 0 1 2
DATA m[10;10], m[1;COL]:=CH(COL*0.1)+SH(COL*0.1)
DATA m[2;COL]:=COL*0.1+TH(COL*0.2)
DATA 1m[15;9], 1m[ROW;COL]:=COL*0.1+SIN(CH(ROW*COL*0.1))
```

Ejemplo IV.2: Inicialización de vectores y matrices.

En el ejemplo anterior, se ha declarado un vector de cinco elementos, y se ha inicializado sus tres primeras componentes a los valores 0, 1 y 2 respectivamente. Las componentes que no se inicializan explícitamente, se inicializan a cero. La siguiente línea declara una matriz de 10x10, y se inicializa la fila uno con la expresión *CH(COL*0.1)+SH(COL*0.1)*, donde la variable *COL* toma los valores de 0 a 9. La última fila muestra la declaración e inicialización de una matriz entera. En este ejemplo, también se pueden observar otras tres extensiones a *CSMP*, se pueden incluir expresiones como parámetros de bloques, y estos además se pueden anidar. Además un identificador puede empezar por dígito.

También se pueden incluir expresiones de inicialización mediante *ROW* y *COL* del ejemplo IV.2 en cualquier sección de un programa *OOCSMP*. Es decir, que las expresiones:

```
m1[ROW;COL]:=COL*0.1+SIN(CH(ROW*COL*0.1))
m[2;COL]:=COL*0.1+TH(COL*0.2)
m[1;COL]:=CH(COL*0.1)+SH(COL*0.1)
```

Ejemplo IV.3: Expresiones con bucle implícito sobre vectores y matrices.

estarían permitidas dentro de cualquier sección. El límite superior de los valores de *ROW* y *COL* cuando aparecen en expresiones dentro de índices, lo marca el tamaño más pequeño de la fila o columna del vector o matriz donde aparece. Cuando *ROW* o *COL* aparecen en una expresión que da lugar a un vector o matriz, se crean antes las variables auxiliares necesarias para que la iteración sea eficiente. Por ejemplo, el siguiente fragmento de código:

```
DATA A[3], B[3], C[3]
...
DYNAMIC
        C[ROW] = A**B
...
```

Ejemplo IV.4: Ejemplo de expresión que genera código optimizado.

en el que se asigna a cada elemento de un vector el producto escalar de otros dos vectores, daría lugar a fragmentar la operación en dos partes:

- Primero se calcula el producto escalar, que se asigna a una variable intermedia auxiliar.
- Luego se realiza la iteración.

También se han implementado las operaciones básicas entre vectores y matrices, que se detallan a continuación.

IV.1.1 Operadores producto, división, suma y resta: *,/,+ y –

Estos operadores entre vectores y matrices representan el producto/división/suma/resta elemento a elemento.

La siguiente tabla muestra los tipos de los operandos y del resultado:

Op. Izdo. \ Op.dcho	Escalar	Vector	Matriz
Escalar	Escalar	Vector	Matriz
Vector	Vector	Vector	Error
Matriz	Matriz	Error	Matriz

Tabla IV.1: Resultado de los operadores *,/,+ y -.

En todas estas operaciones los vectores y/o matrices deben ser de igual dimensión. Si se conoce la dimensión y no es así, se da un mensaje de error en tiempo de compilación[1]. Si no se conoce la dimensión, el error se da en tiempo de ejecución. Ejemplos de operaciones válidas entre matrices son las siguientes:

```
DATA 1m[10;10], 1m[ROW;COL]:=ROW+COL
DATA 2m[10;10], 2m[ROW;COL]:=ROW-COL
DATA res[10;10]
DYNAMIC
  res:=1m+(2m*1m)
```

Ejemplo IV.5: Operaciones con matrices.

IV.1.2 Operador + monádico

Aplicado a vectores/matrices nos devuelve un escalar, que es la suma de todos los elementos del vector/matriz.

IV.1.3 Operador producto matricial: **

Este operador sólo tiene sentido entre vectores o matrices, aplicado a otro tipo de operandos provoca un error, ya sea en tiempo de compilación o de ejecución. Entre vectores representa el producto escalar, los vectores han de ser de igual longitud. Entre matrices representa el producto habitual de matrices, donde el número de columnas de la primera matriz ha de ser igual al número de filas de la segunda. En caso de que los operandos sean un vector y una matriz de una fila, o una columna, el resultado es equivalente al resultado de operar dos vectores.

IV.1.4 Operador módulo: %

Este operador devuelve el resto de dividir el primer operando entre el segundo. Aplicado entre un vector/matriz y un escalar devuelve un vector/matriz cuyos elementos son el resto de la división entre el elemento y el escalar. Aplicado entre vectores o matrices devuelve un vector/matriz al que se ha aplicado la operación elemento a elemento.

La siguiente tabla muestra los tipos de los operandos y del resultado:

[1] Puede ser que no se conozca la dimensión de la matriz/vector si este se pasa como parámetro a un método

Op. izdo. \ Op. dcho.	Escalar	Vector	Matriz
Escalar	Escalar	Error	Error
Vector	Vector	Vector	Error
Matriz	Matriz	Error	Matriz

Tabla IV.2: Resultado del operador %.

■ IV.1.5 Bloques específicos para vectores y matrices

Además de las operaciones anteriores, se han añadido los siguientes bloques, específicos para vectores y matrices:

Nombre	Sintaxis	Definición
Matriz inversa	Mat1 :=INVERSE(Mat2)	Devuelve la matriz inversa de la matriz Mat2; en caso de que la matriz sea singular, se produce un error.
Matriz transpuesta	Mat1 := TRANSPOSE (Mat2)	Devuelve la matriz transpuesta de la matriz Mat2. Si Mat2 es de dimensión NxM, Mat1 debe ser de dimensión MxN.
Determinante	Scalar := DETERMINANT (Mat2)	Devuelve el determinante de la matriz Mat2.

Tabla IV.3: Bloques específicos para vectores y matrices.

■ IV.2 Autosuma, autoresta, autoproducto y autodivisión

En *OOCSMP* se han implementado las operaciones autosuma (+=), autoresta (-=), autoproducto (*=) y autodivisión (/=) que suman o restan el resultado de una expresión a una variable. Se permiten una o varias operaciones de autosuma o autoresta sobre variables, aunque estas ya hayan aparecido en la parte izquierda de una ecuación. Si el resultado de la expresión que aparece a la derecha de la autosuma/resta/producto/división es un vector, se genera una operación para cada una de las componentes del vector. Además, el compilador genera las variables auxiliares intermedias necesarias para que la iteración sea eficiente. Por ejemplo:

```
...
DATA F21[6], F31[6], F32[6], F22[]
DATA F31[] := 1 1 4 13 43 142, F32[] := 1 1 5 17 61 217
DATA F21[] := 1 1 2 3 5 8, F22[] := 1 1 4 10 28 76
DYNAMIC
  A+=(VEC31**VEC32)+(VEC21*VEC41)
  A:=SIN(TIME)
  ...
```

Ejemplo IV.6: Autosuma.

Sería compilado de la siguiente forma:

```
A := SIN(TIME)
AUX1:= PRODUCTO_ESCALAR(VEC31, VEC32)
AUX2:= PRODUCTO_ELEMENTO(VEC21, VEC42)
AUX3:= SUMA(AUX1, AUX2)
para i desde 0 hasta 6
  A := A + AUX3[i]
fin para
```

Como la variable *A* debe tener valor antes de poder aplicarle el operador autosuma, ha de ejecutarse su asignación antes de la autosuma. La operación autosuma se ha dividido en tres fragmentos:

- El primero calcula el producto escalar, y lo asigna a un escalar. En *OOCSMP* esto se hace con una función de la biblioteca.
- El segundo calcula el producto elemento a elemento y lo asigna a un vector auxiliar. En *OOCSMP* esto también se hace con funciones de la biblioteca.
- El tercero genera otro vector auxiliar con el vector resultado de sumar el escalar *AUX1* y el vector *AUX2*. En este caso este paso sería innecesario, ya que, el resultado del bucle anterior es el mismo que el de:

```
para i desde 0 hasta 6
  A := A + AUX1 + AUX2[i]
```

  Ya que la autosuma de la suma de un vector y un escalar es igual a la autosuma del vector y el escalar. Es decir que: *A=AUTOSUMA(SUMA(ESCALAR, VECTOR))* es equivalente a *A= AUTOSUMA(ESCALAR, VECTOR)*. Esta última optimización, mediante la aplicación de reglas de simplificación de expresiones, no se ha implementado en *OOCSMP*.

- El bucle de la autosuma propiamente dicho.

Obviamente el tiempo de ejecución del fragmento de código (complejidad O(n)) anterior es mucho menor que el tiempo de ejecutar el código no optimizado, que sería de complejidad $O(n^2)$:

```
A := SIN(TIME)
para i desde 0 hasta 6
  A := A + SUMA( PRODUCTO_ESCALAR(VEC31, VEC32),
                 PRODUCTO_ELEMENTO(VEC21, VEC42))[i]
fin para
```

ya que dentro del bucle de la autosuma estaríamos realizando innecesariamente un producto escalar y un producto elemento a elemento que son constantes en todas las pasadas del bucle.

IV.3 Expresiones lógicas

Se han introducido los operadores lógicos =, # (distinto), <, >, <=, >=, || (o lógico) y && (y lógico) para la construcción de expresiones lógicas. Dichas expresiones pueden ser utilizadas dentro de la construcción *START()* de las primitivas multimedia, o como condición de la instrucción *FINISH*.

IV.4 Manejo de eventos

Se han sobrecargado los bloques *INSW* y *FNCSW* para el manejo de eventos. Cuando estos bloques no aparecen en la parte derecha de una ecuación, se comportan de forma parecida a las construcciones *SI...ENTONCES* y *SI...ENTONCES...SI NO* de la programación tradicional.

La sintaxis de los dos bloques es la siguiente:

Bloque	Descripción
INSW(X1,EQ1,EQ2)	if (X1<=0) then EQ1 else EQ2
FCNSW(X1,EQ1,EQ2,EQ3)	if (X1<0) then EQ1 else if (X1==0) then EQ2 else EQ3

Tabla IV.4: Sintaxis de los bloques manejadores de eventos.

El bloque *INSW* ha sido modificado cuando se utiliza como manejador de eventos (cuando se utiliza como bloque su semántica es *X1<0*) para facilitar su uso con variables de estado o discretas en las que conviene discriminar el valor 0 de los valores positivos.

Donde *EQ1*, *EQ2* y *EQ3* pueden ser cualquier ecuación de *OOCSMP*. También se permite el anidamiento de estos bloques.

Si en las ecuaciones *EQ1*, *EQ2* o *EQ3*, se modifica alguna variable involucrada en una integral, se reinicia el método de integración, calculándose un nuevo "valor inicial" para la expresión de la integral. Esto es una mejora de otros esquemas de tratar las discontinuidades [Broe96] (en los que siempre que se produce un evento de estado se reinician todos los integradores) debido a que el primer paso en un integrador numérico siempre es de menor (como mucho igual) precisión que los siguientes

Por ejemplo, si tenemos el siguiente fragmento de código:

```
V := INTGRL (V0, X+X0)
INSW ( X-POS, X*=2, )
```

Ejemplo IV.7: Discontinuidad.

Cuando *X<=POS*, se producirá una discontinuidad en la variable *X*, por tanto, se reinicializará el método de integración para esta variable, empezando a calcular el primer valor con el método rectangular, con valor "inicial" igual a *X*2*. Esto es necesario para poder captar la discontinuidad en la integral, ya que si el método de integración siguiese usando la información de puntos obtenidos antes de producirse la discontinuidad, el resultado sería incorrecto. Si *X>POS*, no se ejecuta nada.

■ IV.5 Sobrecarga de bloques

Todos los bloques de *CSMP* se han extendido, de forma que como parámetros admiten expresiones (que pueden estar compuestas por bloques). El resultado de estas expresiones puede ser un escalar, un vector o una matriz. Las siguientes tablas muestran las posibles combinaciones de los tipos de los parámetros de los bloques.

Resultado	Parámetro
Escalar	Escalar
Vector	Vector
Matriz	Matriz

Tabla IV.5: Tipos del parámetro y resultado de los bloques aritméticos (ver tabla III.3)

Sintaxis	Tipos de los parámetros			
Y=STEP(X)	Y	X		
	Escalar	Escalar		
	Vector	Vector		
	Matriz	Matriz		
Y=RAMP(X)	Y	X		
	Escalar	Escalar		
	Vector	Vector		
	Matriz	Matriz		
Y=IMPULS(P1,P2)	Y	P1	P2	
	Escalar	Escalar	Escalar	
	Vector	Vector	Vector	
	Vector	Escalar	Vector	
	Vector	Vector	Escalar	
	Matriz	Matriz	Matriz	
Y=SAMPLE(P1,P2,P3)	Y	P1	P2	P3
	Escalar	Escalar	Escalar	Escalar
	Vector	Escalar	Escalar	Vector
	Vector	Escalar	Vector	Escalar
	Vector	Escalar	Vector	Vector
	Vector	Vector	Escalar	Escalar
	Vector	Vector	Escalar	Vector
	Vector	Vector	Vector	Escalar
	Vector	Vector	Vector	Vector
	Matriz	Matriz	Matriz	Matriz

Y = PULSE (P, X)	Y	P	X
	Escalar	Escalar	Escalar
	Vector	Vector	Vector
	Vector	Escalar	Vector
	Vector	Vector	Escalar
	Matriz	Matriz	Matriz

Y = SAWTH (P1, P2)	Y	P1	P2
	Escalar	Escalar	Escalar
	Vector	Vector	Vector
	Vector	Escalar	Vector
	Vector	Vector	Escalar
	Matriz	Matriz	Matriz

Y = SINE(P1,P2,P3)	Y	P1	P2	P3
	Escalar	Escalar	Escalar	Escalar
	Vector	Escalar	Escalar	Vector
	Vector	Escalar	Vector	Escalar
	Vector	Escalar	Vector	Vector
	Vector	Vector	Escalar	Escalar
	Vector	Vector	Escalar	Vector
	Vector	Vector	Vector	Escalar
	Vector	Vector	Vector	Vector
	Matriz	Matriz	Matriz	Matriz

Y = RNDGEN(P)	Y	P
	Escalar	se ignora
	Vector	se ignora
	Matriz	se ignora

Y = GAUSS(P1, P2)	Y	P	X
	Escalar	Escalar	Escalar
	Vector	Vector	Vector
	Vector	Escalar	Vector
	Vector	Vector	Escalar
	Matriz	Matriz	Matriz

Tabla IV.6: Tipos de parámetros y resultados de los generadores de ondas

Además, al igual que con el manejo de eventos discretos, si hay una discontinuidad en un bloque integral, provocada por los bloques discontinuos: *STEP*, *IMPULS*, *SAMPLE*, *PULSE*, se reinicia el integrador asociado a esa integral.

Sintaxis	Tipo de los parámetros			
Y = LIMIT(P1,P2,X)	Y	P1	P2	X
	Escalar	Escalar	Escalar	Escalar
	Vector	Escalar	Escalar	Vector
	Vector	Escalar	Vector	Escalar
	Vector	Escalar	Vector	Vector
	Vector	Vector	Escalar	Escalar
	Vector	Vector	Escalar	Vector
	Vector	Vector	Vector	Escalar
	Vector	Vector	Vector	Vector
	Matriz	Matriz	Matriz	Matriz

Y:=DEADSP(P1;P2;X)	Y	P1	P2	X
	Escalar	Escalar	Escalar	Escalar
	Vector	Escalar	Escalar	Vector
	Vector	Escalar	Vector	Escalar
	Vector	Escalar	Vector	Vector
	Vector	Vector	Escalar	Escalar
	Vector	Vector	Escalar	Vector
	Vector	Vector	Vector	Escalar
	Vector	Vector	Vector	Vector
	Matriz	Matriz	Matriz	Matriz
Y:=QNTZR(P;X)	Y	P	X	
	Escalar	Escalar	Escalar	
	Vector	Vector	Vector	
	Vector	Escalar	Vector	
	Vector	Vector	Escalar	
	Matriz	Matriz	Matriz	
Y:=HSTRSS(IC;P1;P2;X)	Si ((P1=Escalar) Y (P2=Escalar) Y (X=Escalar) Y (IC=Escalar)) entonces Y:=Escalar si no Si ((P1=Matriz) Y (P2=Matriz) Y (X=Matriz) Y (IC=Matriz)) entonces Y:=Matriz si no Y:=Vector			

Tabla IV.7: Tipos de parámetros y resultados de los modificadores de ondas

IV.6 Bloques de usuario

En *OOCSMP*, se pueden definir nuevos bloques, que coinciden con la noción de procedimientos y funciones de la programación tradicional, si bien las ecuaciones incluidas en un bloque, por defecto, también se reordenan. Los bloques definidos por el usuario pueden o no devolver un valor. Para devolver un valor, dentro del bloque se ha de asignar el valor a un variable con el mismo nombre que el bloque (de forma semejante a como se hace en el lenguaje *Pascal*), el tipo que devuelve el bloque es el tipo que resulta de evaluar la parte derecha de esta igualdad. Por el momento, no está permitido que el tipo retorno sea un objeto o una colección de objetos. Esta asignación no tiene porqué ser la última ecuación del bloque, se reordena con el resto de las ecuaciones. El valor que devuelve un bloque de usuario puede o no recogerse en una asignación, tal como ocurre en lenguajes como *C* o *Java*. La recursividad está permitida.

En *OOCSMP* no hay variable locales a bloques.

Los parámetros escalares se pasan por valor, el resto se pasan por referencia, y se pueden modificar. Los bloques se declaran indicando su nombre, y a continuación los parámetros, separados por comas. La sintaxis en la que se declaran los parámetros formales de un bloque es:

Tipo	Sintaxis
Escalar	<nombre-parámetro>
Vector	<nombre-parámetro>[ ]
Matriz	<nombre-parámetro>[;]
Objeto	<nombre-clase> <nombre-objeto>
Colección de objetos	<nombre-clase> <nombre-objeto> [ ]

Tabla IV.8: Formato de los parámetros formales.

Si un bloque espera un objeto como parámetro, y se le invoca con una colección de objetos, se produce una iteración, ejecutándose el bloque con cada uno de los elementos de la colección como parámetro. Algo parecido sucede si el bloque espera un dato escalar como parámetro, y se le invoca con un vector. En este caso, también se produce una iteración sobre cada uno de los elementos del vector. Si

un bloque no recibe parámetros se puede llamar con o sin paréntesis abiertos y cerrados. La ventaja de esta última forma de invocar las funciones se ve más claramente en la orientación a objetos, en el apartado IV.2.5.

■ IV.7 Ejemplo: El juego de la vida

Como ejemplo de las extensiones básicas que se han añadido a *OOCSMP*, y como prueba de que, aunque *OOCSMP* es un lenguaje de simulación continua, se puede realizar también simulación discreta, se va a presentar un ejemplo de una simulación discreta de uno de los autómatas celulares más famosos: el juego de la vida [Gayl95] [Wolf94]. Este juego fue el precursor de los llamados sistemas de vida artificial (*a-life systems*) que son de gran interés hoy, no sólo por sus implicaciones biológicas sino por el desarrollo de agentes inteligentes

El juego se desarrolla en una malla bidimensional (en nuestro caso una matriz 20x20). Vamos a considerar condiciones de contorno absorbentes (es decir, que consideramos que los elementos de la primera fila y de la última fila no son vecinos, tampoco los elementos de la primera y última columna). Un elemento de la malla con un valor de uno se dice que está vivo, y si el elemento tiene el valor cero, se dice que está muerto. El sistema evoluciona aplicando una serie de reglas a todas las casillas simultáneamente. Las reglas son las siguientes:

- Un elemento vivo con dos vecinos vivos permanece vivo.
- Un elemento con tres vecinos vivos bien permanece vivo, o si no lo estaba, nace.
- El resto de los elementos muere o permanece muerto.

Como vecinos de un elemento, se tomarán los 8 vecinos más cercanos (vecindad de *Moore*).

En nuestro modelo hemos creado un bloque que cuenta los vecinos de un elemento genérico, teniendo en cuenta las condiciones de contorno absorbentes. Se han usado los bloques que manejan eventos para la comprobación de las condiciones de borde. Las reglas se aplican en un bucle implícito sobre todos los elementos de la matriz. Para simular la aplicación de las reglas simultáneamente sobre todos los elementos, se usa una matriz auxiliar. El modelo *OOCSMP* queda como sigue:

```
DATA DX:=20, DY:=20
DATA M[DY;DX], MDF[DY;DX]
DATA M[ROW;COL]:=0, MDF[ROW;COL]:=0
DATA M[5;5] := 1, M[5;6] := 1, M[5;7] := 1, M[6;7] := 1, M[7;6] := 1

NEIGHBORS F, C, MAT[;]
  INSW ( F, NN:=0, NN:=MAT[F-1;C] )
  INSW ( F, NN+=0, INSW( C,       ,NN+= MAT[F-1;C-1] ) )
  INSW ( F, NN+=0, INSW( DX-C-1, ,NN+= MAT[F-1;C+1] ) )

  INSW ( DY-F-1, NN+=0, NN+= MAT[F+1;C] )
  INSW ( DY-F-1, NN+=0, INSW( C, ,NN+= MAT[F+1;C-1] ) )
  INSW ( DY-F-1, NN+=0, INSW( DX-C-1, ,NN+=MAT[F+1;C+1]) )

  INSW ( C, , NN+=MAT[F;C-1] )
  INSW ( DX-C-1, , NN+=MAT[F;C+1] )

  NEIGHBORS := NN

DYNAMIC
  NOSORT
  MDF[ROW;COL] := FCNSW ( 2-NEIGHBORS(ROW,COL,M),
                          FCNSW ( 3-NEIGHBORS(ROW,COL,M), 0, 1, 0),
                          M[ROW;COL],
                         0 )
  M[ROW;COL] := MDF[ROW;COL]
TIMER FINTIM := 100, delta:=1
ISOPLOT [C], 0, 0, 1, 1, M
```

Listado IV.1: Modelo del juego de la vida.

Se ha configurado la matriz con un conjunto de elementos (que se conocen como *glider*) que repiten periódicamente su estructura, pero desplazada. Un momento en la simulación del modelo anterior se muestra en la figura:

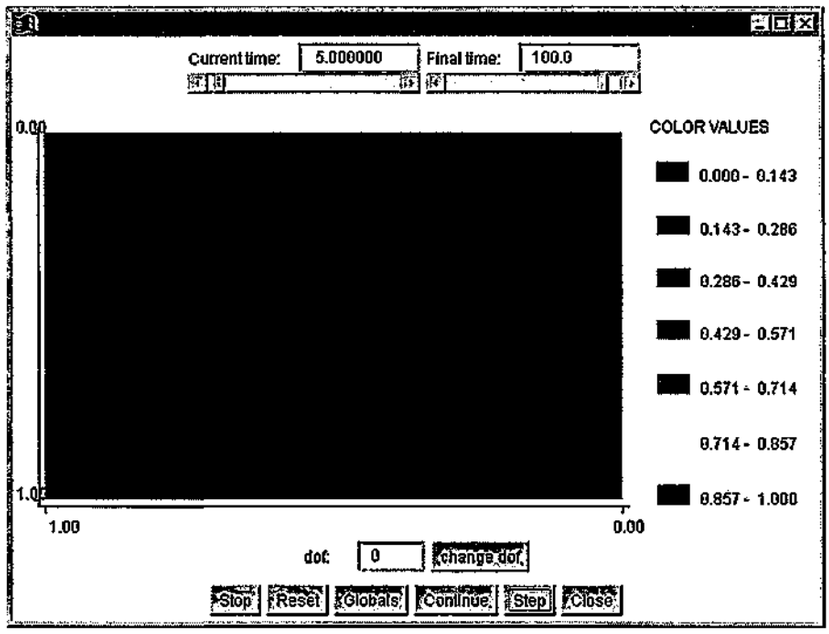


Figura IV.1: Un momento de la ejecución del autómata celular.

Como se puede observar, de color más oscuro aparecen los elementos vivos, mientras que de color más claro aparecen los elementos muertos. Durante la ejecución de la simulación podemos cambiar el estado del juego, cambiando el valor de los elementos de la matriz *M* (accesible desde el botón *Globals*, ver sección IX.4.1). Se ha compilado con la opción `-step` (ver sección IX.1). Esta opción genera un botón (*step*) que permite ejecutar la simulación paso a paso, para ver la evolución del sistema.

IV.8 Extensiones orientadas a objetos

IV.8.1 Introducción a la Orientación a Objetos

El término "orientado a objetos" (OO) se acuñó en los años sesenta, e intenta organizar el desarrollo de un sistema de una forma más natural que con la programación tradicional. Se trata de una evolución hacia un paradigma de programación más cercano a nuestra forma de pensar. Es el más estructurado, modular y racional de los intentos que se han hecho en el campo de la abstracción de datos hasta hoy. Para ver el porqué, se van a comparar los distintos tipos de programación [Sier99].

Un programa consta siempre de dos partes: las instrucciones ejecutables (o programa en sentido estricto) y los datos sobre los que actúan las instrucciones. Según cómo se organicen datos y programas, se distinguen las siguientes formas de programar:

- Programación procedimental o clásica. Las instrucciones se ejecutan secuencialmente, siendo la unidad básica de ejecución el *programa*. Cada programa puede invocar o llamar a otros por medio de la instrucción *CALL* o equivalente, por lo que todos los que forman una aplicación se integran en una jerarquía de llamadas. Uno de los programas, situado en la raíz de la jerarquía, recibe el nombre de *programa principal*. Los demás se llaman *funciones*, *subrutinas*, *subprogramas* o *procedimientos*. Los datos no tienen organización predefinida, son auxiliares de los programas y proporcionan valores que sirven para realizar cálculos o para decidir la secuencia de ejecución de las instrucciones.

- Programación lógica. Las instrucciones no tienen un orden de ejecución prefijado, sino que reciben control de un *procesador de inferencias*, que en cada momento decide el orden en que ha de ejecutarse (actuar o desencadenarse) cada una de ellas. Las instrucciones se llaman *reglas*. Los datos están desordenados y se reducen a casos particulares de las reglas, en las que faltan las premisas.

- Programación orientada a objetos, *OOP* (del inglés *Object Oriented Programming*). La programación orientada a objetos enfatiza en los datos, al contrario que la programación estructurada que enfatiza en los programas. Los programas no forman jerarquías ni se invocan directamente y son los datos (más bien los *objetos*) los que constituyen la jerarquía básica. Un objeto puede estar ligado a otro a través de una relación, lo que da lugar a la aparición de una red (un *diagrama de objetos*) que sustituye a la jerarquía que forman los programas en las aplicaciones procedimentales. También en *OOP* existen programas, pero son meras componentes de los objetos, y su ejecución no se desencadena con la instrucción *CALL*, sino por medio de un *mensaje* que alguien (el usuario o un programa asociado al mismo o a otro objeto) envía a un objeto determinado. El receptor del mensaje decide qué programa debe ejecutarse. En lenguajes orientados a objetos *puros* ésta es la única forma de invocar la ejecución de un programa. Pero también hay otros lenguajes *híbridos* que toleran la instrucción *CALL* y en los que se puede mezclar la programación orientada a objetos con la programación procedimental clásica. Este es el caso de *OOCSMP*.

La *OOP* organiza los datos de un programa al igual que los objetos del mundo real. Por ejemplo, los departamentos de una compañía (ventas, personal, contabilidad, etc.), los diversos artículos que vende una tienda (discos, libros, películas, etc.); todos ellos serían objetos.

Los elementos fundamentales de la programación orientada a objetos son tres:

- **Objetos**: Son tipos estructurados que combinan datos (*propiedades* o *atributos*) y comportamiento (programas encargados de tratar dichos datos: *métodos*).

- **Clases**: Son conjuntos de objetos con información y comportamientos similares. Es una plantilla que define los atributos y los métodos. Todos los objetos son especializaciones de una clase. Todos los objetos de la misma clase usan el mismo método (código) en respuesta a una petición de servicio específica. Por ejemplo, una clase puede describir las características fundamentales de un libro que se

vende en una tienda (título, autor, año de edición, número de páginas, *PVP*, etc.), mientras que un objeto representará un libro específico (El extranjero, Albert Camus, 131, 1200, etc).

- **Herencia**: las clases de objetos pueden heredar información y comportamientos de otras clases. Existen dos clases de herencia:

  - *Herencia simple:* una clase puede ser generalizada por una sola superclase. (Una superclase, en cambio, puede tener varias subclases). Siguiendo con el ejemplo anterior, si la tienda también vende periódicos, podemos crear una clase *publicación* que generalice a la clase *libro* y a la *periódico*, La clase *publicación* tendría como atributos el título, fecha de edición, número de páginas, *PVP*, etc. La clase *libro* heredaría todas estas propiedades, y además añadiría el autor, el tipo de libro, el *ISBN*, el año de edición, etc. *periódico*, por su parte, añadiría el número de orden, la fecha de edición, el tipo de periódico, etc. Este tipo de herencia es la que se ha implementado en *OOCSMP*.

  - *Herencia múltiple*, que se aplica cuando una clase puede ser generalizada por dos o más superclases independientes. La implementación de la herencia múltiple es más compleja que la herencia simple y presenta ciertos problemas, por lo que algunos lenguajes (como Smalltalk y Java) no permiten realizarla. La principal ventaja de la herencia múltiple es la posibilidad de definir clases híbridas, que comparten las propiedades de dos o más clases predefinidas, sin tener que escribir código o reduciendo éste al mínimo. Su inconveniente principal es que resulta más difícil de implementar, pues no es fácil saber de dónde se hereda una propiedad o un método, además de que pueden producirse incompatibilidades y ambigüedades, especialmente si una clase hereda de otra por dos caminos diferentes. Este tipo de herencia no ha sido implementada en *OOCSMP*.

Los objetos se declaran invocando un método especial, que se llama *constructor del objeto*. Este método se utiliza para inicializar los atributos y realiza otras acciones iniciales. El nombre del constructor coincide con el nombre de la clase a la que pertenece el objeto. Cuando un objeto no se va a utilizar más, se invoca a otro método especial, que se llama *destructor*. En *OOCSMP* se han implementado los constructores pero no los destructores.

En 1987, P. Wegner clasificó los lenguajes y sistemas informáticos en tres grupos, dependiendo de cuáles de los elementos anteriores permiten utilizar. Esta clasificación es estándar en la nomenclatura informática:

- **Basados en objetos**: incorporan el concepto de *objeto*.

- **Basados en clases**: Además, agrupan los objetos en *clases* semejantes entre sí, que se caracterizan porque los mismos programas y propiedades se aplican a todos los objetos de la misma clase, pero no a los de clases diferentes.

- **Orientados a objetos:** Además de objetos y clases, permiten la herencia. Este es el caso del *OOCSMP*.

En la programación orientada a objetos, los atributos de los objetos no se pueden acceder directamente. El acceso a los atributos lo proporcionan los métodos adecuados (la interfaz del objeto). Esto define el primer principio de la orientación a objetos: *el encapsulamiento*. Un objeto es una caja negra que no permite distinguir los detalles de lo que hay en su interior. Esta propiedad de los objetos se denomina también *ocultación de datos*. Para acceder a los atributos de un objeto, algunos de los métodos o programas disponibles para cada clase de objetos deben ser públicos, y su conjunto constituye la *interfaz* de los objetos de esa clase.

Los objetos se relacionan entre sí por medio de *mensajes*. Un mensaje consta de tres partes: objeto al que se dirige, método que se invoca y argumentos. Puede ser que el receptor de un mensaje no se conozca hasta el tiempo de ejecución, es decir, hay un enlace tardío (*late binding*) entre el mensaje (servicio requerido) y el método (fragmento de código) [Lee97].

La figura IV.2 muestra un diagrama en el que se representan varios objetos (formados por datos, encapsulados por métodos) comunicándose mediante el paso de mensajes.

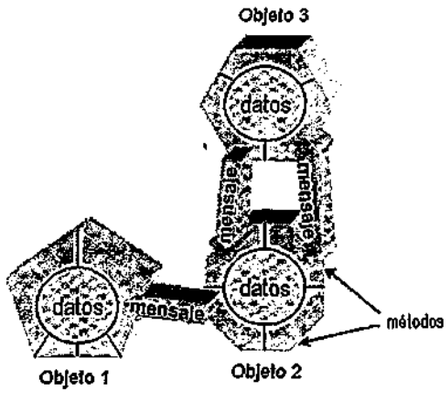


Figura IV.2: Paso de mensajes entre objetos, encapsulamiento.

Otra característica primordial para que pueda hablarse de orientación a objetos es el *polimorfismo*, que permite que los datos y programas asociados a una clase de objetos puedan tener el mismo nombre que los de otra clase diferente. Objetos diferentes reaccionan de modo diferente al mismo mensaje. Así por ejemplo, en *OOCSMP*, podemos aplicar los métodos *, /, +, -, etc. tanto a escalares como a vectores o matrices. Esta forma de polimorfismo, que se aplica a métodos definidos sobre la misma clase, recibe el nombre de *sobrecarga* y exige que haya alguna diferencia entre los argumentos de ambas versiones, ya sea en su número o en sus tipos.

Así pues, un objeto puede considerarse formado por tres partes:

- *relaciones*: están formadas por referencias o puntero a otros objetos. Hay tres tipos de relaciones:
  - *de herencia*: especifican que una clase de objetos es subclase de otra más general.
  - *de pertenencia o de agregación*: indican que un objeto es parte de otro.
  - *de asociación*: especifican cualquier otra relación entre dos objetos.

- *propiedades o atributos*: son datos encapsulados dentro del objeto. Si la propiedad se aplica a la clase, tiene valor único, común para todos los objetos de la clase, llamamos a la propiedad *variable de clase*. Si se aplica cada uno de los objetos se llama *variable de objeto*, y cada objeto de la clase tiene su propio valor. Respecto a la herencia, las propiedades pueden ser *propias* (definidas en la clase a la que pertenece el objeto) o *heredadas* (definidas en una clase diferente, de la que hereda la clase a la que pertenece el objeto).

- *métodos*: son las operaciones que pueden realizarse sobre el objeto, normalmente están expresadas en forma de programas (código) que el objeto es capaz de ejecutar y que también pone a disposición de sus descendientes a través de la herencia. Al igual que las propiedades pueden ser propios o heredados.

Otra técnica usada frecuentemente en *OOP* es la *delegación*, el trabajo va (vía pase de mensajes) de un objeto (cliente) a otro (agente) porque desde la perspectiva del cliente, el agente tiene los servicios que el cliente necesita. El trabajo se pasa constantemente hasta que llega al objeto que contiene los datos y los métodos para realizar el trabajo.

Por tanto, la programación tiene algunas ventajas sobre la programación tradicional, como son:

- **Facilidad de análisis y diseño**: La orientación a objetos está más cercana a nuestra forma de pensar que la programación tradicional.

- **Modularidad**: Un programa es modular si se compone de partes independientes y robustas. Un programa modular se presta con más facilidad a la reutilización de sus componentes, así como a la verificación o depuración de las mismas. En un sistema orientado a objetos, la modularidad es más

intuitiva, pues las clases de objetos son módulos naturales, que corresponden a una imagen lógica de la realidad.

- **Extensibilidad**: La extensibilidad hace de la programación orientada a objetos una técnica de programación incremental. Un modelo bien construido permitirá, sin complicaciones excesivas, añadir:
  - Nuevos objetos pertenecientes a las clases preexistentes.
  - Nuevas propiedades a dichas clases.
  - Conductas o métodos nuevos.
  - Nuevas clases y subclases.

- **Eliminación de redundancias**: la herencia evita la definición múltiple de propiedades y métodos comunes a muchos objetos.

- **Reutilización**: La programación orientada a objetos proporciona un marco perfecto para la reutilización de las clases. El encapsulamiento facilita el acoplamiento de las mismas clases en aplicaciones distintas, pues el aislamiento que les proporciona significa que es posible añadir una clase o un módulo nuevo (extensibilidad) sin afectar al resto de la aplicación. La facilidad de reutilización es uno de los motivos principales que explican la proliferación de la metodología orientada a objetos. Los sistemas comerciales de *OOP* vienen provistos de un conjunto de clases predefinidas, lo que permite ahorrar tiempo y esfuerzos en el desarrollo de las aplicaciones. Tal es el caso de la biblioteca de componentes estándar que hemos desarrollado para *OOCSMP*, y que se detalla en el apéndice A.

IV.8.2 Clases *OOCSMP*

Las clases en *OOCSMP* se corresponden con la noción clásica de clase de los lenguajes de programación orientados a objeto. Es decir, representan un conjunto de elementos con propiedades similares. Dentro de la declaración de la clase se pueden incluir todas las secciones de un programa *CSMP*. En *OOCSMP sólo* se ha implementado la herencia simple. El esquema de una clase *OOCSMP* se muestra en la siguiente figura:

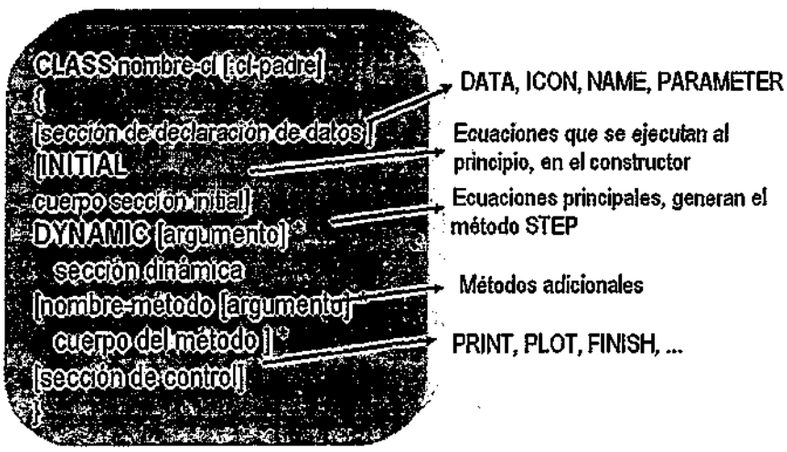


Figura IV.3: Esquema de una clase *OOCSMP*.

El siguiente es un ejemplo de la declaración de una clase.

```
CLASS BALL
{
        DATA posX:=1.0, posY:=2.0, radius:=1.0
        ...
}
```

Ejemplo IV.8: Declaración de una clase.

Todos los parámetros de tipo escalar declarados en la sección *DATA* de una clase que no son inicializados son parámetros obligatorios del constructor. Los parámetros declarados en *DATA*, que además se inicialicen, son parámetros optativos. En este ejemplo, el constructor de la clase podría tener cero, uno, dos o tres parámetros, ya que *posX, posY* y *radius* tienen valor. Los vectores y matrices, como se han de declarar siempre en una sección *DATA* (ya que se ha de indicar su dimensión), son siempre optativos, se inicialicen o no. Los objetos declarados dentro de una construcción *DATA* son obligatorios, los declarados fuera no se pueden pasar como parámetros. En el constructor del objeto se dan los valores por defecto a los atributos, y se asigna el valor de los parámetros a los atributos correspondientes, también se ejecutan las instrucciones incluidas en la sección *INITIAL*.

La herencia se declara mediante el operador ':'. Si en la clase hija se extiende un método de la clase padre, antes de la ejecución del método de la clase hija, se ejecuta el de la clase padre. Se pueden incluir tantos métodos como se quiera dentro de una clase. Un método termina donde empieza el siguiente, o bien comienza otra sección *OOCSMP*, tal como definición de la salida (instrucciones *PLOT, ICONICPLOT, PLOT3D*, etc.), declaración de condiciones de finalización, etc.

Si dentro de la definición de una clase se incluye alguna instrucción de control del tipo *PRINT* o *PLOT* (ver sección VI), se dibujan o se imprimen los atributos especificados de todos los objetos que se declaren de la clase, en el mismo panel.

También se puede declarar una clase, pero dejarla indefinida, añadiendo a continuación del nombre de la clase un ';'. Se podrán entonces referenciar objetos de esta clase, por ejemplo como parámetros de métodos.

Dentro de una clase se pueden declarar atributos de dos tipos especiales: *ICON* y *NAME*. Su utilidad se presenta a continuación.

La sección *DYNAMIC* de una clase se invoca mediante el método *STEP*.

IV.8.2.1 El tipo *ICON*

Las variables de tipo *ICON* se han de declarar dentro de una clase, y toman como valor una cadena de caracteres delimitada por comillas dobles ("). Esta cadena de caracteres especifica un nombre de fichero de tipo *gif*, que será asociado con el objeto correspondiente. El fichero podrá estar en la máquina local, o bien, si el lenguaje objeto seleccionado es Java, en un servidor Web. Este icono podrá ser usado en las distintas representaciones gráficas que se elija (ver apartado VI). Esta variable sólo puede tomar valor en el constructor del objeto, o bien cuando se declara. No se puede modificar en el programa, aunque sí en tiempo de ejecución, haciendo uso de la ventana emergente para el cambio de variables. El siguiente ejemplo muestra la definición de una variable de tipo *ICON*.

```
CLASS BALL
{
        ICON icon:= "ball.gif"
        DATA posX:=1.0, posY:=2.0, radius:=1.0
        ...
}
```

Ejemplo IV.9: Declaración e inicialización de un atributo de tipo *ICON*.

IV.8.2.2 El tipo *NAME*

Las variables de tipo *NAME* se han de declarar dentro de una clase, y toman como valor una cadena de caracteres delimitada por comillas dobles ("). Esta cadena de caracteres especifica el nombre del objeto que aparecerá en el botón de la interfaz de usuario (tanto Java como C++) que generará el compilador *C-OOL*. Esta variable sólo puede tomar valor en el constructor del objeto. El siguiente ejemplo muestra la definición de una variable de tipo *NAME*.

```
CLASS BALL
{
        NAME name
        ICON icon:= "ball.gif"
        DATA posX:=1.0, posY:=2.0, radius:=1.0
        ...
}
```

Ejemplo IV.10: Declaración de un atributo de tipo *NAME*

■ IV.8.3 Objetos *OOCSMP*

Los objetos *OOCSMP* se pueden declarar en la lista de los parámetros que vienen a continuación de la instrucción *DATA*, o bien independientemente. Por ejemplo, supongamos que queremos declarar un objeto de la clase *BALL*, del ejemplo IV.1, las siguientes declaraciones son válidas:

```
DATA BALL b0("b0")
BALL b1("b1","http://ii.uam.es/~jlara/ball.gif")
BALL b3("ball-3", "ball.gif", 4.0, b2.posX)
DATA BALL b2("ball-2", "ball.gif", 3.0*SIN(EXP(2.7)))
```

Ejemplo IV.11: Declaración de objetos.

La declaración del objeto *b1* sobreescribe el valor por defecto del atributo de tipo *ICON*. La declaración de *b2* está sobreescribiendo el valor por defecto del atributo *posX*, que es 1, al valor resultado de la expresión `3.0*SIN(EXP(2.7))`. La declaración de *b3* también sobreescribe este valor, así como el de *posY*. Esta última declaración también muestra la forma de acceder a los atributos del objeto, mediante la sintaxis: `<nombre-objeto>.<nombre-atributo>`.

La declaración de los objetos también se reordena, para que estas se ejecuten en el orden correcto. Así en el ejemplo anterior, la declaración de *b2* se llevaría a cabo antes que la declaración de *b3*, ya que el constructor de *b3* contiene una referencia a un atributo de *b2*.

Como ya se dijo, la diferencia entre declarar un objeto dentro o fuera de la instrucción `DATA`, es que si estamos dentro de la definición de una clase, y el objeto está dentro de `DATA`, puede ser pasado como parámetro de la clase. Si el objeto se declara fuera de `DATA`, no se puede pasar como parámetro.

■ IV.8.4 Colecciones de objetos

Los objetos se pueden agrupar en colecciones. Todos los objetos de una colección han de ser de la misma clase. Las colecciones, al igual que los objetos, se pueden declarar dentro de una instrucción `DATA`, o fuera. La diferencia entre declarar la colección dentro o fuera de `DATA` es la misma que para los objetos. El siguiente ejemplo define dos colecciones con los objetos del ejemplo IV.2.

```
DATA BALL myArray1 := b1 b2 b3
BALL myArray2 := b4 b5 b0
```

Ejemplo IV.12: Declaración de colecciones de objetos.

El compilador *implícitamente* declara un *array* con todos los elementos que pertenecen a una misma clase. Este *array* tiene el mismo nombre que la clase. Esto nos va a facilitar, por ejemplo, el añadir objetos a la simulación de forma dinámica en tiempo de ejecución, o invocar métodos sobre todos los elementos de una clase.

■ IV.8.5 Métodos

Dentro de una clase, se pueden declarar tantos métodos como se quiera, con sintaxis igual que la de declaración de bloques de usuario (sección IV.1.6). La declaración de un método termina cuando comienza una sección o un nuevo método, termina la clase, comienza la declaración del formato de salida, o una condición especial de finalización (instrucción `FINISH`)

Los métodos se pueden invocar sobre un objeto, sobre una colección de objetos o sobre una clase. Si se invocan sobre una colección de objetos, el método se ejecuta sobre cada elemento de la colección. Si se invoca sobre una clase, se ejecuta sobre todos los elementos de la clase. Las posibles sintaxis de invocación de un método son las siguientes:

```
<objeto>.<metodo>([<param1> (,<param2>)*])
<clase>.<metodo> ([<param1> (,<param2>)*])
<coleccion>.<metodo> ([<param1> (,<param2>)*])
```

Sintaxis IV.3: Invocación de métodos.

Si un método espera un objeto como parámetro, y se le invoca con una colección de objetos, se produce una iteración, ejecutándose el método con cada uno de los elementos de la colección como parámetro. Si el receptor del método es una colección de objetos o una clase, entonces se ejecuta el

método siempre que el objeto receptor no coincida con el objeto parámetro. Esto nos es útil en muchas ocasiones, por ejemplo, en el modelo de la gravitación, los planetas tienen interacciones gravitatorias con el resto, pero no consigo mismos, en el modelo del listado IV.2, las bolas pueden colisionar con el resto de bolas, pero no consigo mismas, etc.

Algo parecido sucede si el método espera un dato escalar como parámetro y se le invoca con un vector. En este caso, también se produce una iteración sobre cada uno de los elementos del vector.

Al igual que los bloques de usuario (sección IV.1.10), los valores de retorno se especifican asignándolos a una variable con el mismo nombre que el método. Si un método no devuelve nada, no se especifica esta asignación. La sección *DYNAMIC* de una clase (método *STEP*) no puede devolver ningún valor.

La sección *DYNAMIC* dentro de una clase, se puede declarar con parámetros, como cualquier otro método. Esta sección se puede invocar mediante las siguientes sintaxis:

```
<objeto>.STEP([<param1> (,<param2>)*])
<clase>.STEP([<param1> (,<param2>)*])
<coleccion>.STEP([<param1> (,<param2>)*])
```

Sintaxis IV.4: Invocación de la sección *DYNAMIC*.

Si un método no lleva parámetros, se puede invocar con dos paréntesis, abierto y cerrado, y sin nada dentro, o bien sin paréntesis. Invocar un método sin paréntesis tiene la ventaja de que no hay que preocuparse de si un cierto valor está implementado como una función o bien como un atributo. El ejemplo típico es el de una clase coordenada, que tuviera atributos con formato rectangular o polar, y funciones para convertir de rectangular a polar o viceversa. Con esta forma de invocar los métodos, da igual si los atributos son las coordenadas rectangulares y las polares se calculan mediante un método, o es al contrario. La clase se podría cambiar y el resto del modelo no se tendría que cambiar en absoluto.

El siguiente es un ejemplo de declaración e invocación de varios métodos basándonos en la clase *BALL* del ejemplo IV.10. Para mayor claridad, se ha simplificado el modelo, de forma que aunque se permiten bolas de masas diferentes, cuando se produce una colisión, las bolas salen con una velocidad igual pero de sentido contrario a la que llevaban. Las velocidades se suponen constantes,

```
TITLE Balls
DATA EPS := 0.001
CLASS BALL
{
  NAME name
  ICON icname := "ball.gif"
  DATA pos0[2], pos0[]:=0 0, radius:=1.0, Mass:=1.0
  DATA v[2], v[]:=0 1, pos[2], distv[2]
  DYNAMIC
    pos := INTGRL(pos0,v)
  COLLIDE BALL b
    distv := b.pos-pos
    dist  := SQRT(distv[0]*distv[0]+distv[1]*distv[1])
    x     := dist-radius-b.radius
    INSW(-(EPS-x)*(x-EPS), INSW(col, v:= -v, ), col:=-1)
    INSW(-(EPS-x)*(x-EPS), INSW(col, col:=1, ), )
  PLOT [C], 3, 40, -3, -25, pos[1], pos[0]
  PLOT [E], pos[1], TIME
  PRINT pos[1]
}
DATA p1[2], p1[] := 0 10, v1[2], v1[] := 0 -2
BALL b0("b0")
BALL b1("b1","ball1.gif",p1,0.5,2.0,v1)
DYNAMIC
  BALL.STEP()
  BALL.COLLIDE(BALL)
TIMER FINTIM:=25.0, delta:=0.1, PLdelta:=0.2, PRdelta:=0.2
```

Listado IV.2: Declaración e invocación de métodos.

El ejemplo anterior contiene dos invocaciones a métodos:

- La primera es *BALL.STEP()*, es decir, se llama a la sección principal (*DYNAMIC*) de todos los objetos declarados de la clase *BALL*.
- La segunda es *BALL.COLLIDE(BALL)*, es decir, se llama al método *COLLIDE* de todos los objetos de la clase *BALL*. Como el argumento es un vector, y en la declaración del método se ha indicado que debería ser un objeto, se realiza un doble bucle. Además, como el vector receptor del mensaje es el mismo que el argumento, se llamaría al método con todos los objetos que no sean el objeto receptor. Al producirse una colisión, se produce una discontinuidad en la velocidad, que como está siendo integrada, hace que se reinicie el integrador asociado, tomándos como nueva condición 'inicial' la nueva velocidad.

Dentro de la clase *BALL* se declaran tres salidas gráficas (ver capítulo VI), las tres representan gráficamente ciertos atributos de todos los objetos de la clase *BALL*. El resultado de compilar el modelo anterior con *C-OOL*, y de ejecutar el mismo se muestra en la figura V.4.

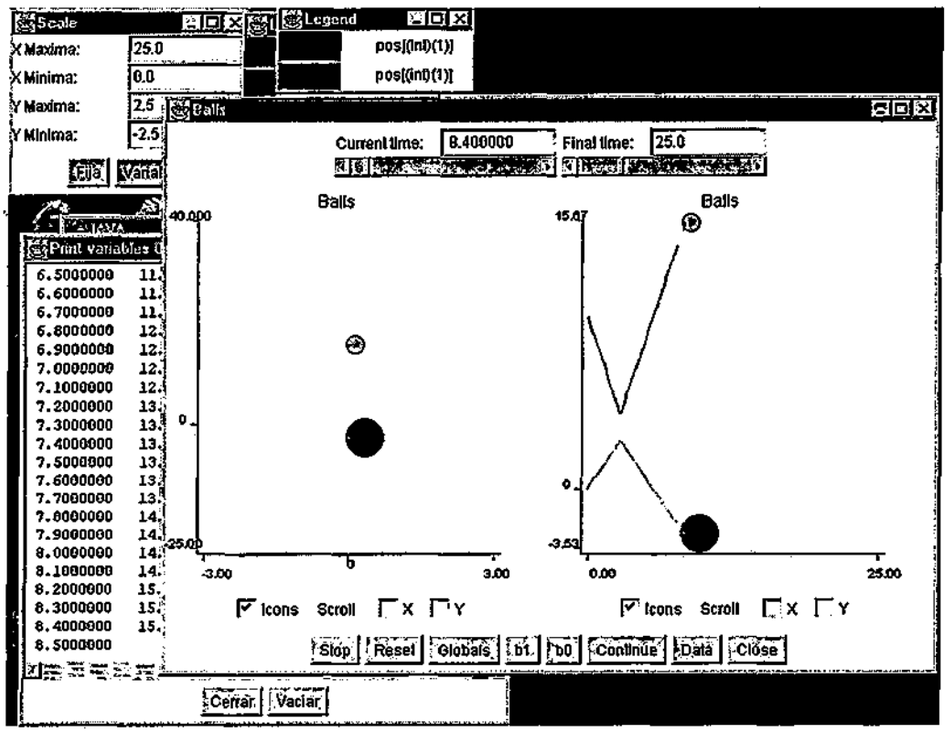


Figura IV.4: Ejecución del modelo del listado anterior.

IV.9 Otras extensiones

A continuación se detallan otras extensiones que también se han añadido al lenguaje.

IV.9.1 Inclusión de ficheros

En el punto del programa donde aparezca la instrucción *INCLUDE* se inserta el contenido del fichero que se indique. Lo habitual es que el fichero contenga la definición de una o varias clases que de esta forma pueden ser reutilizadas, aunque el fichero puede contener código *OOCSMP* de cualquier clase. Se admiten instrucciones *INCLUDE* anidadas, y varias instrucciones *INCLUDE* dentro de un mismo fichero.

■ IV.9.2 La instrucción *PARAMETER*

Si después de haber declarado un dato de tipo escalar mediante *DATA*, luego se especifica en la instrucción *PARAMETER*, este dato podrá ser leído como parámetro del programa. Si no se hace así, tomará el valor que se le haya asignado en *DATA*. Por ejemplo:

```
DATA SunMass :=332999
....
PARAMETER SunMass, b4.radius
```

Ejemplo IV.13: Declaración de parámetros de la simulación.

En este ejemplo, se intentarían leer como parámetros tanto la variable global *SunMass* como el atributo *radius* del objeto *b4*, si no se especifican como entrada al programa, tomarían el valor de la instrucción *DATA*. Si, una vez compilado a C++, el programa se llama, por ejemplo, *BALLS.EXE*, los parámetros se indicarían de la siguiente forma:

```
BALLS SunMass= 32000 b4.radius= 2.5
```

Ejemplo IV.14: Paso de parámetros a la simulación en C++.

En el caso de que se esté generando un *applet*, el paso de parámetros para el ejemplo anterior, se realizaría de la siguiente forma:

```
<applet code = "balls.class">        <param name= SunMass value=32000>
                                     <param name= b4.radius value=2.5>
```

Ejemplo IV.15: Paso de parámetros a la simulación en un *applet*.

Si bien es posible describir una página *HTML* con las instrucciones *SODA* del capítulo VIII y la opción *-HTMLpage= <nombre>* del compilador *C-OOL* (ver sección IX).

■ IV.9.3 La instrucción '\'

Especifica que a continuación podemos añadir instrucciones que modifiquen los datos de la simulación que acabamos de definir. Es decir, al compilar un programa a Java que contenga esta instrucción se crearan botones 'especiales', uno para ejecutar la simulación 'principal', y el resto para ejecutar la simulación con las modificaciones indicadas. Esto es útil cuando se quieren sugerir algunos ejemplos interesantes al usuario, o bien cuando se quiere cambiar el contenido de alguna salida gráfica, para mostrar otras variables de la simulación (ver el ejemplo de la sabana africana, en el que se crean varios botones para ver algunas de las cadenas tróficas).

Las instrucciones permitidas tras '\' son:

- *TIMER*, *DATA* y *TITLE*.
- Asignaciones a variables, que se ejecutan una vez (no se reordenan con el código de la simulación principal), al ser pulsado el botón correspondiente.
- Instrucciones de declaración de formatos de salida (*PRINT*, *PLOT*, *ISOPLOT*, etc.).
- Creación de objetos
- Asignación de objetos a colecciones de objetos.

El siguiente ejemplo presenta el problema de un cuerpo orbitando alrededor de otro, más precisamente de la tierra girando alrededor del sol. En la simulación principal, los parámetros están ajustados para que la órbita sea un círculo. Se han diseñado otras simulaciones alternativas, en las que se han modificado los parámetros para que la órbita sea elíptica, parabólica, hiperbólica, o bien una caída libre. El listado es el siguiente:

```
TITLE GRAVITATION
DATA G:=0.00011869, PI:=3.141592653589793
DATA MS:=332999
INCLUDE "Planet.csm"
* Actual planets
```

```
Planet Earth  ("Earth",1,        0,      1, -6.287,0.107, 0    )
Earth.STEP()
* Time intervals and other data                                          *
TIMER delta:=.0005, FINTIM:=2, PRdelta:=.1, PLdelta:=.01
PLOT Earth.Y, Earth.X
METHOD ADAMS
\
TITLE ELIPSE
TIMER FINTIM:=100, PLdelta:=.05
DATA Earth.XP0:=-8.5
PLOT 3.2, 2.5, -4.5, -11
\
TITLE ELIPSE
TIMER FINTIM:=100, PLdelta:=.1
DATA Earth.XP0:=-8.8
PLOT 8.0 , 2.5, -8.7 , -57.31
\
TITLE PARABOLA
TIMER FINTIM:=75, PLdelta:=.1
DATA Earth.XP0:=-8.9
PLOT 2.5 , 2.5, -120.0 , -26.0
\
TITLE HIPERBOLA
TIMER FINTIM:=75, PLdelta:=.1
DATA Earth.XP0:=-9
PLOT 2.5 , 2.5, -140.0 , -57.0
\
TITLE CAIDA LIBRE
TIMER PLdelta:=.001
DATA Earth.XP0:=0
PLOT 0.5 , 1.5, -0.5 , -0.5
```

Listado IV.3: Uso de la instrucción '\'.

Lo que sucede al compilar a Java el ejemplo anterior es que se generan seis botones, el primero para ejecutar la simulación principal, y los siguientes para ejecutar cada una de las simulaciones. Como se puede observar, en la instrucción *PLOT* se especificar sólo un cambio en la escala.

Este modelo se ha usado en la primera página del curso de gravitación (ver apartado X.2), el *applet* resultante se muestra en la figura IV.5.

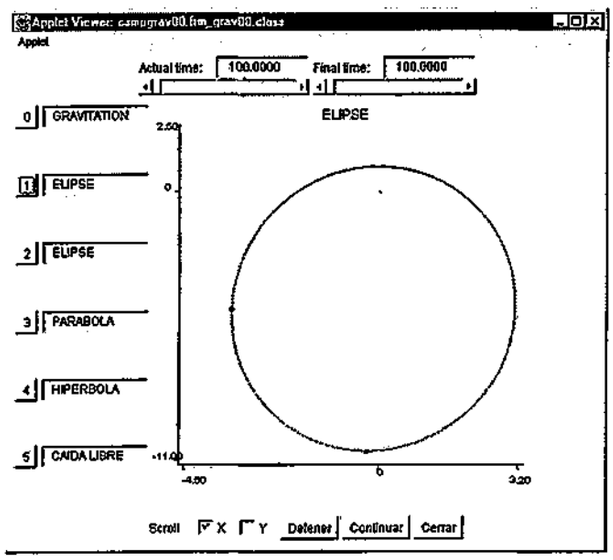


Figura IV.5: Ejemplo de uso de la instrucción '\'.

V. Ecuaciones en derivadas parciales

En esta sección se van a presentar las extensiones que se han añadido a *OOCSMP* para la resolución de ecuaciones y sistemas de ecuaciones en derivadas parciales (*PDEs*).

Comienza el capítulo con una breve introducción a las *PDEs*, a los sistemas existentes de resolución de *PDEs*, y a las capacidades de *OOCSMP* para resolverlas.

A continuación, en las secciones 2, 3 y 4, se describen las distintas primitivas para la construcción de los dominios, las primitivas para discretizarlos, seguido de una presentación de los distintos métodos de solución de *PDEs* implementados.

En la sección 5 se describe cómo solucionar sistemas de ecuaciones, y ecuaciones cuasilineales. Como ejemplo, en esta sección se resuelve la ecuación del seno de Gordon.

En la sección 6, se presentan algunos ejemplos de resolución de *PDEs*, algunos de ellos, muestran las ventajas de usar la tecnología de objetos al resolverlas.

Por último se presenta la herramienta *MGEN*. Esta herramienta está construida en Java, y permite el diseño gráfico de dominios, la discretización de los mismos y la imposición de condiciones iniciales y de contorno. Puede ser usada de dos formas:

- Como un programa Java independiente. Esta forma de uso nos permite la generación de código *OOCSMP* de las mallas diseñadas, así como la lectura de código *OOCSMP* de dominios y mallas, para su modificación.
- Como parte de una simulación. De esta forma, se puede modificar el dominio, la malla o las condiciones durante la ejecución de la simulación.

V.1 Introducción

V.1.1 Ecuaciones en derivadas parciales.

Como ya se ha comentado, las ecuaciones en derivadas parciales (*PDEs*) gobiernan multitud de fenómenos físicos, tales como aquellos de dinámica de fluidos, campos electromagnéticos, procesos químicos, etc. Estas *PDEs* se suelen dividir en tres grupos, dependiendo de sus curvas características, o curvas de propagación de información. Expresamos una *PDE* general de segundo orden como:

$$a(\cdot)u_{xx} + 2b(\cdot)u_{xy} + c(\cdot)u_{yy} + d(\cdot)u_x + e(\cdot)u_y + f(\cdot)u + g(\cdot) = 0$$

Ecuación V.1: Forma general de una *PDE*.

Donde u_x significa la derivada parcial de u respecto a x, u_{xx} significa la derivada segunda de u respecto a x^2, etc. También se suelen designar por, $\frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}$ etc.

Las funciones $a(\cdot)$, pueden depender de x, y, t, u, etc.

Podemos clasificar a las *PDEs* como:

- Hiperbólicas (cuando $b^2\text{-}ac > 0$), el ejemplo típico es la ecuación de ondas.

$$\frac{\partial^2 u}{\partial t^2} = c^2 \Delta u$$

Ecuación V.2: Ecuación de ondas.

Donde Δ es el operador Laplaciano, que viene dado por $\partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$

Esta ecuación describe el comportamiento del sólido elástico, de las ondas de sonido en el aire, de las ondas electromagnéticas, de los seismos propagándose en la tierra, etc.

- Parabólicas (cuando $b^2\text{-}ac = 0$), por ejemplo, la ecuación de difusión, que viene dada por la expresión:

$$\frac{\partial u}{\partial t} = k \Delta u$$

Ecuación V.3: Ecuación de difusión.

Esta ecuación describe la conducción del calor, el movimiento browniano, las dinámicas de poblaciones, etc.

- Elípticas (cuando $b^2\text{-}ac < 0$), por ejemplo, la ecuación de Poisson, que viene dada por la expresión:

$$\Delta u = f(\vec{x})$$

Ecuación V.4: Ecuación de Poisson.

Esta ecuación gobierna fenómenos electrostáticos, magnetostáticos, etc.

El tipo de la ecuación gobierna el número y naturaleza de las condiciones de contorno que se han de especificar para que la solución sea única, y esté bien determinada [Hild68].

Los dos primeros tipos de ecuaciones, hiperbólicas y parabólicas, definen problemas de valor inicial (o Cauchy): Si la información de la función que queremos calcular se da en el instante inicial (también se ha de indicar alguna condición de contorno, ver figura V.1), entonces las ecuaciones describen cómo la ecuación se propaga con el tiempo, es decir, los dos primeros tipos describen una evolución temporal. El término "inicial", se refiere, por tanto a que una de las variables independientes es el tiempo.

Sin embargo, el tercer tipo define un problema de valores de contorno. Porque lo que se quiere es encontrar una función estática, independiente del tiempo, que satisface la ecuación dentro del dominio de interés, con algún tipo de restricción en el contorno.

Por tanto, es más importante, desde el punto de vista de la resolución computacional distinguir el tercer tipo de los dos primeros, que los dos primeros entre sí. Además muchos problemas son mezcla de los dos primeros tipos.

Si una ecuación es de tipo hiperbólico en una cierta región, existe una cierta familia de curvas en el plano *xy* (llamadas curvas características). Estas curvas "transportan" el valor de la condición de contorno a través del dominio.

También podemos clasificar las *PDEs* por el orden de la derivada más alta que aparece en la ecuación, o bien por la linealidad. En este último caso, podemos distinguir:

- Ecuaciones lineales, las funciones que multiplican a los términos de la función solución no contienen la propia función, p.e.: $u_x+bu_y = 0$.
- Ecuaciones cuasilineales, cuando los operadores que multiplican a los términos de la función solución pueden ser la función solución, pero no una derivada parcial, p.e: $u_x+uu_y = 0$.
- Ecuaciones no lineales, cuando los operadores que multiplican a los términos de la función solución pueden ser la función solución, o una derivada parcial. p.e.: $u_x+(u_y)^2= 0$.

En cuanto a las condiciones de contorno, se pueden clasificar en tres tipos [Stra92]:

- Condiciones de tipo *Dirichlet*, si se especifica la función que buscamos en el contorno. Es decir, se expresa de la forma:

$$u = g(\vec{x}, t)$$

Ecuación V.5: Condición de tipo Dirichlet

- Condiciones de tipo *Neumann*, si se especifica la derivada normal de la función.

$$\frac{\partial u}{\partial n} = g(\vec{x}, t)$$

Ecuación V.6: Condición de tipo Neumann

- Condiciones de tipo *Robin*, si la condición se expresa de la forma:

$$\frac{\partial u}{\partial n} + a(\vec{x}, t)u = g(\vec{x}, t)$$

Ecuación V.7: Condición de tipo Robin

Cualquiera de estas condiciones se llama homogéneas en caso de que la función *g* sea igual a cero.

Normalmente, las ecuaciones hiperbólicas se asocian con condiciones de tipo Cauchy en una región abierta, las parabólicas se asocian con condiciones de tipo Dirichlet o Neumann también en una región abierta, y las elípticas con condiciones Dirichlet o Neumann en una región cerrada [Lapi82].

Las siguientes tres figuras muestran diagramas de problemas hiperbólicos, parabólicos y elípticos. Las líneas discontinuas son las curvas características de la ecuación (curvas en las que la solución de la ecuación permanece constante). El borde inferior de la figura (*t=0*) representa una condición inicial, los bordes izquierdo y derecho representan condiciones de borde, asumiendo que la dimensión x es finita.

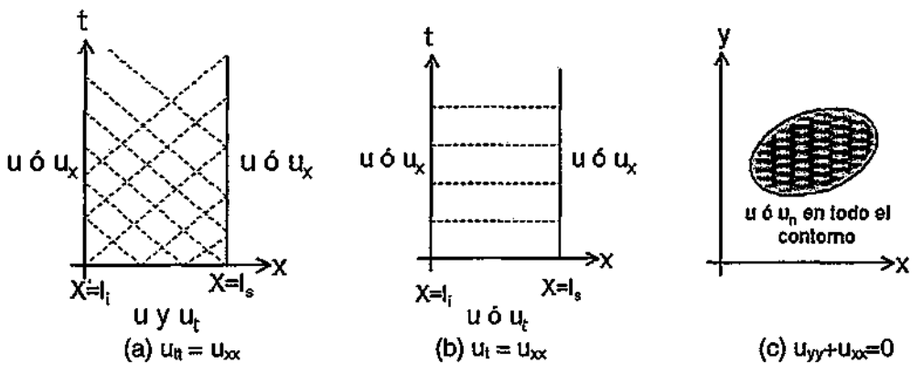


Figura V.1a,b,c: Representación de las condiciones impuestas a ecuaciones hiperbólicas (a), parabólicas (b) y elípticas (c). Las líneas discontinuas son las curvas características.

Otra forma de clasificación de las condiciones, que es la que se ha adoptado para el presente trabajo, es la de condiciones *naturales*, y condiciones *esenciales*. Las condiciones *esenciales* son equivalentes a las condiciones de *Dirichlet*. Las condiciones *naturales* especifican un flujo. Por ejemplo, en la ecuación del calor en una dimensión [Otto92]:

$$\frac{\partial T}{\partial t} + \frac{\partial}{\partial x}\left(Ak \frac{\partial T}{\partial x} \right) + Q = 0$$

Ecuación V.8: Ecuación del calor

Donde T es la temperatura, A es el área de una sección del material, k es la conductividad térmica, Q es el calor que se proporciona al material. La condición de contorno natural, viene dada por el flujo de calor, que es:

$$q = -k \frac{\partial T}{\partial x}$$

Ecuación V.9: Condición natural

En el contorno del dominio en el que queremos solucionar la ecuación, habrá zonas donde especificaremos condiciones de contorno esenciales, y otras en las que especificaremos condiciones de contorno naturales.

Las condiciones naturales, nos vienen dadas al reformular la *PDE* en forma "débil". Esta formulación se utiliza al resolver la *PDE* mediante el método de los elementos finitos.

Para simular dominios infinitos, se utilizan las condiciones de contorno periódicas, en las que el valor de un lado del dominio tiene el mismo valor que el lado opuesto.

Los métodos analíticos de resolución de *PDEs* (separación de variables, expansiones en series de Fourier, etc. [Stra92]) son limitados: hay ecuaciones que no se pueden resolver analíticamente. Es necesario entonces el empleo de métodos numéricos de resolución. Para resolver la *PDE* mediante métodos numéricos, es necesario discretizar el dominio donde vamos a resolver la ecuación, de forma que la ecuación se calculará en un número discreto de puntos del interior del dominio. Algunos de los métodos numéricos más empleados son:

- **Diferencias finitas** [Stri89]: Consiste en sustituir cada derivada por una discretización de la misma. Las discretizaciones serán series de Taylor truncadas. Hay muchos tipos de esquemas, dependiendo de la discretización que elijamos para cada derivada. Además pueden quedar esquemas explícitos - en los cuales no hay que resolver sistemas de ecuaciones, sólo recorrer convenientemente los nodos de la malla – o bien métodos implícitos, en los que para solucionar cada fila de la malla, hay que plantear un sistema de ecuaciones. Algunos de los métodos explícitos más conocidos, son el esquema clásico, el de DuFort-Frankel, el de Richardson, el de Lax-Wendroff, etc. Entre los implícitos están el de Crank-Nicolson y el *ADI*.

La discretización que se debe llevar a cabo para este método es mediante cuadriláteros paralelos a los ejes *X* e *Y*. Normalmente los cuadriláteros serán de igual tamaño (figura V.2).

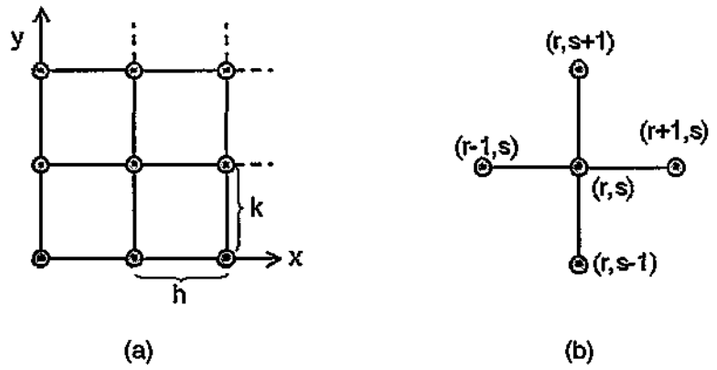


Figura V.2a,b: Malla para diferencias finitas (a), indexación de los nodos de la malla (b).

Por ejemplo, una expansión en serie de Taylor para u(x,y), en el punto (r,s) es:

$$u(x_r+h,y_s)=u(x_r,y_s)+hu_x\Big|_{(r,s)}+\frac{h^2}{2!}u_{xx}\Big|_{(r,s)}+\frac{h^3}{3!}u_{xxx}\Big|_{(r,s)}+\ldots$$

Ecuación V.10: Desarrollo de Taylor para u(x,y) en el punto (r,s)

con un error de $O(h^4)$. Reordenando la ecuación, obtenemos el valor de la derivada:

$$u_x\Big|_{(r,s)}=\frac{u(x_r+h,y_s)-u(x_r,y_s)}{h}-\frac{h}{2!}u_{xx}\Big|_{(r,s)}-\frac{h^2}{3!}u_{xxx}\Big|_{(r,s)}-\ldots$$

Ecuación V.11: Valor de $u_x(x,y)$ en el punto (r,s)

Truncando, y conservando sólo el primer término, obtenemos:

$$u_x\Big|_{(r,s)}\approx\frac{u(x_r+h,y_s)-u(x_r,y_s)}{h},$$

$$Er=\pm\frac{h}{2}u_{xx}\Big|_{(\xi,s)}=O(h);x_r\le\xi\le x_r+h,x_r-h\le\xi\le x_r$$

Ecuación V.12: Valor aproximado $u_x(x,y)$ en el punto (r,s)

El esquema anterior se conoce como esquema hacia adelante ("forwards"). Desarrollando en serie de Taylor $u(x_r-h,y_s)$, se obtiene el esquema de hacia atrás ("backwards"). Restando el primero del segundo, se obtienen las diferencias centradas (ver tabla V.4). Los esquemas para la segunda derivada se obtienen de forma parecida.

Si un esquema en diferencias finitas sólo necesita información en la fila n para obtener el valor de la función en la fila $n+1$, se llaman de un paso. Los que necesita información de varias filas, se llaman multipaso. Un esquema multipaso, de m pasos, necesita los valores de la función solución en los $m-1$ primeros niveles, o deben calcularse mediante otro método [Stri89].

Si la solución aproximada que calcula un esquema converge a la solución verdadera de la ecuación, cuando el espaciado de la malla tiende a cero, el esquema se llama convergente. Un esquema es estable si los errores generados en la computación, tales como los de redondeo y truncamiento, se disipan cuando el cálculo avanza en la malla. Un esquema es consistente si los errores de truncamiento locales obtenidos al discretizar en serie de Taylor tienden a cero cuando h, k, y el paso elemental de tiempo tienden a cero. El error de discretización es una combinación del error de truncamiento de la ecuación y los errores de las condiciones iniciales y de contorno [Shih83].

Los métodos explícitos tienen una desventaja, y es que no se puede avanzar en el tiempo todo lo deprisa que se quiera: el tamaño del paso de tiempo está restringido por el valor de h en la malla. Esta relación la da la condición de Courant-Friedrichs-Lewy (*CFL*) [Cour28], que para el caso de la ecuación $u_t = au_x$, nos da la relación: $|a\Delta t/\Delta x| \leq 1$. Si no se cumple esta restricción, el esquema se vuelve inestable. El sentido geométrico de esta relación queda reflejado en la figura V.3 ($\lambda = \Delta t/\Delta x$). Esta condición no se aplica a los esquemas implícitos. Así, podemos hablar de esquemas incondicionalmente estables, condicionalmente estables e inestables.

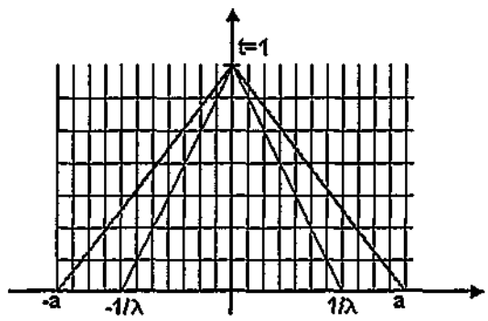


Figura V.3: Sentido geométrico de la condición *CFL*.

- **Elementos Finitos** [Zien89]: Consiste en aproximar la ecuación solución en porciones pequeñas del dominio, llamados elementos finitos o símplices [Otto92]. Estos elementos no tienen por qué ser cuadriláteros, como en el caso de las diferencias finitas. Pueden ser triángulos, cuadriláteros de ocho o nueve nodos, etc. La función incógnita u se representa en cada elemento por polinomio interpolante que es continuo con su derivada hasta un determinado orden. Una vez que se determina la aproximación en cada elemento, la solución para el dominio entero se obtiene ensamblando la solución obtenida en cada elemento.

Para la obtención de los polinomios interpolantes, en el método de los elementos finitos, se usan las funciones de forma ($N(x,y)$). Hay una función de forma por cada vértice del símplice. $N_i(x,y)$ vale uno en el nodo i, y cero en el resto de los nodos (figura V.4).

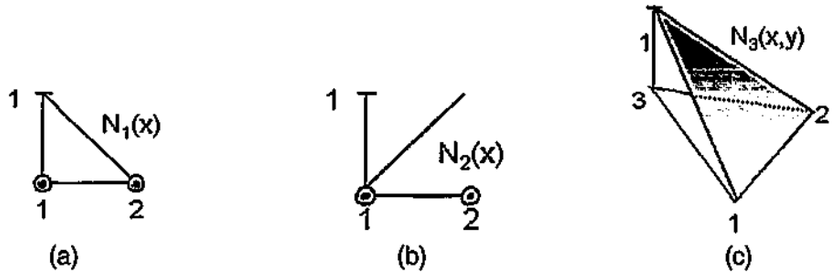


Figura V.4a,b,c: Funciones de forma, para el caso del elemento lineal unidimensional (a) y (b), y una de las funciones de forma del elemento triángulo lineal (c).

La función incógnita se discretiza en cada elemento utilizando estas funciones de forma, dando lugar a la expresión:

$$u(\bullet) \approx \hat{u}(\bullet) = \sum_{i=1}^{n} N_i^{(e)} u_i$$

Ecuación V.13: Discretización de la función incógnita.

Para la obtención de la formulación de elementos finitos, vamos a usar el enfoque del método de los residuos (*"weighted residuals"*, *MWR*), en particular el método de Galerkin. Hay otros enfoques, pero llevan a la misma formulación.

Al sustituir la función u por la aproximación, dejan de satisfacerse tanto las ecuaciones como las condiciones de contorno. A ese error se le llama residuo. El objetivo del método es minimizar este residuo. Podemos expresamos la *PDE* que queremos resolver como:

$$A(u) = Lu(\bullet) - p = 0 \qquad \text{en } \Omega \text{ (dominio)}$$

$$B(u) = Mu(\bullet) - r = 0 \qquad \text{en } \Gamma \text{ (contorno)}$$

Ecuación V.14: Formulación de la *PDE* a resolver.

Donde L y M son operadores diferenciales, y p y r funciones. Al sustituir u por su discretización, obtenemos:

$$L\hat{u}(\bullet) - p = R(\bullet)_\Omega$$
$$M\hat{u}(\bullet) - r = R(\bullet)_\Gamma$$

Ecuación V.15: Residuos en el interior del dominio y en el contorno al discretizar la ecuación

La formulación *MWR*, minimiza el residuo mediante la expresión:

$$\int_\Omega W_i R_\Omega d\Omega + \int_\Gamma \overline{W}_i R_\Gamma d\Gamma = 0$$

Ecuación V.16: Minimización del residuo

en cada elemento, para i desde 1 hasta el número de nodos del elemento. El método de Galerkin consiste en elegir las funciones de peso (W_i) como las funciones de forma N_i.

Una vez que se ha obtenido el sistema de ecuaciones anterior para cada elemento del dominio, se ha de proceder al ensamblado, en el cual a partir de m (número de elementos del dominio) sistemas de n (número de nodos del elemento) ecuaciones del tipo: $K^{(e)} u = f^{(e)}$, se llega a una ecuación $Ku=f$, que se obtiene sumando las contribuciones los elementos de $K^{(e)}$ y $f^{(e)}$ que correspondan al mismo nodo en K y f. Así, el elemento (i,j) de la matriz K, tendrá contribuciones de elementos de todas las matrices $K^{(e)}$ que contengan al nodo (i,j). La matriz K se llama comúnmente matriz de rigidez.

- **Elementos de contorno (boundary elements):** Este método es similar a los elementos finitos, pero sólo el contorno del dominio necesita ser discretizado. La ventaja que se obtiene es que la dimensión del problema se reduce en uno. Este método se usa principalmente en problemas de acústica, análisis de tensiones, flujo de potencial, mecánica con fractura, etc. [Kirk99].

- **Elementos discretos:** Este método está orientado a problemas formados por un gran número de cuerpos moviéndose. Los problemas de este tipo no se pueden solucionar por medio de elementos finitos, ya que éstos están orientados a problemas continuos, no con este tipo de

discontinuidades. El método se usa para determinar la topología del contacto dinámico de los cuerpos.

- **Autómatas celulares** [deCo97]: Algunos fenómenos físicos se pueden simular mediante autómatas celulares. Por ejemplo, la ecuación de difusión, se puede modelar mediante un autómata celular que en cada nodo de la malla tenga un valor que corresponda a la concentración, o a la temperatura. En cada iteración, el valor de un nodo se reemplaza por la media de los valores de sus ocho vecinos de la vecindad de tipo *Moore*.

- Hay otros muchos métodos, como el de los volúmenes finitos, métodos estocásticos, métodos basados en analogías eléctricas [deCo97] (métodos *TLM*), etc.

V.1.2 Generación de mallas

El primer paso para resolver una *PDE* mediante elementos finitos o diferencias finitas es discretizar el dominio. Hay muy diversas formas de hacerlo. Se suelen clasificar en métodos estructurados [Thom85], [Knup93] y no estructurados [Geor91]. Las mallas estructuradas, tienen una conectividad regular de los puntos, es decir, cada punto tiene el mismo número de vecinos. Las mallas no estructuradas tienen una conectividad irregular, cada punto puede tener un número distinto de vecinos [Fili96]. Entre los métodos estructurados, se encuentran:

- Generación de mallas algebraica. Mediante interpolación. Consiste en pasar del dominio físico al computacional (un cuadrado [-1,1]x[-1,1]), mediante una proyección que sea uno a uno, y en la que el contorno del espacio computacional debe proyectarse en el contorno del espacio físico. La expresión para esta proyección se obtiene mediante interpolación. Una técnica para la proyección desde ciertos dominios al dominio computacional es la de los elementos isoparamétricos, en la que la expresión para la interpolación se obtiene mediante las funciones de forma usadas en el método de los elementos finitos [Hugh87]. Es decir, para pasar del plano *XY* (físico) al $\xi\eta$ (computacional) y viceversa, vamos a usar las funciones de forma *Ni* (para *i* desde 1 al número de nodos del elemento). Por ejemplo, para transformar el cuadrado [-1,1] al rectángulo curvilíneo de 8 nodos, vamos a usar las expresiones:

$$X(\xi,\eta)=\sum_{i=1}^{8} N_i(\xi,\eta)\cdot X_i,\ Y(\xi,\eta)=\sum_{i=1}^{8} N_i(\xi,\eta)\cdot Y_i$$

Ecuación V.17: Expresión para pasar del dominio computacional al físico, para el caso del elemento de 8 nodos.

donde *(Xi, Yi)* es la coordenada del vértice i-ésimo del rectángulo curvilíneo. Las funciones de forma del elemento rectángulo de 8 nodos, vienen dadas por:

$$N_1(\xi,\eta)=-1/4(1-\xi)(1-\eta)(\xi+\eta+1)\ ;\quad N_2(\xi,\eta)=1/2(1-\eta)(1-\xi^2)$$
$$N_3(\xi,\eta)=1/4(1+\xi)(1-\eta)(\xi-\eta-1)\ ;\quad N_4(\xi,\eta)=1/2(1+\xi)(1-\eta^2)$$
$$N_5(\xi,\eta)=1/4(1+\xi)(1+\eta)(\xi+\eta-1)\ ;\quad N_6(\xi,\eta)=1/2(1+\eta)(1-\xi^2)$$
$$N_7(\xi,\eta)=1/4(1-\xi)(1+\eta)(-\xi+\eta-1)\ ;\quad N_8(\xi,\eta)=1/2(1-\eta^2)(1-\xi)$$

Ecuación V.18: Funciones de forma para el rectángulo de 8 nodos.

- Generación de mallas mediante la resolución de *PDEs*. Consiste en definir una *PDE* para las coordenadas del espacio físico en términos de las coordenadas del espacio computacional, y resolver esas ecuaciones en una malla en el espacio computacional para crear la malla en el espacio físico. Como ecuación se usa principalmente *PDEs* de tipo elíptico, que son apropiadas cuando el dominio es cerrado. Las alternativas son ecuaciones hiperbólicas y parabólicas, que se usan para dominios no cerrados.

  - Generación elíptica de mallas [Thom85]. Este método resuelve una *PDE* elíptica para las coordenadas computacionales ξ^1 y ξ^2, en término de las coordenadas físicas x^1 y x^2. El ejemplo más simple es la ecuación de Laplace:

$$\Delta \xi^{i} = 0$$

Ecuación V.19: Ecuación de Laplace

Los bordes del dominio físico se van a corresponder con los bordes del dominio computacional, en el que las coordenadas van a permanecer constantes.

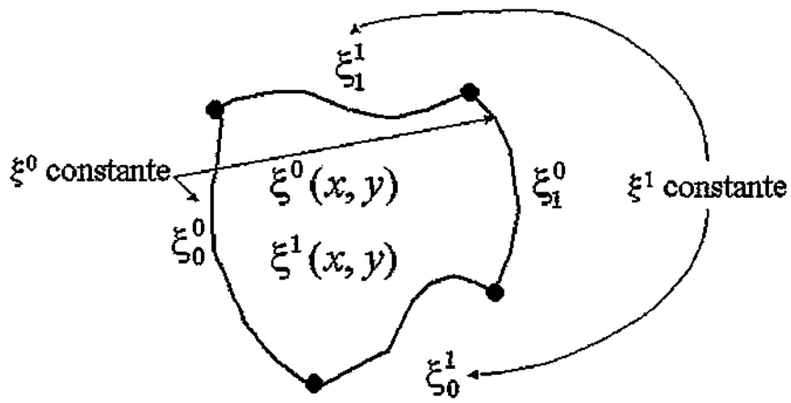


Figura V.5: Bordes en la generación elíptica de mallas (adaptada de [Fili96]).

Esta ecuación elíptica, nos garantiza que el máximo y el mínimo de las dos coordenadas las vamos a encontrar en los bordes, ya que las ecuaciones elípticas siguen el principio del máximo y el mínimo [Lapi82].

Para el control del espaciado de la malla y de la pendiente, podemos añadir una función de control a la ecuación de Laplace, dando como resultado la ecuación de *Poisson* (ecuación V.4). La función de control se correspondería con la función *f* de la ecuación V.4, variando la función *f*, podemos conseguir atractores a cierta línea de coordenada, a un cierto punto particular del dominio y cambios en el espaciado de las líneas o en la inclinación. Estos efectos también pueden conseguirse variando los coeficientes de la ecuación de Laplace, por ejemplo, resolviendo sistemas del tipo:

$$\nabla(D\nabla\xi^{i}) = 0$$

Ecuación V.20: Sistema para generadores elípticos

D sirve como función de peso, la suavidad de la malla se enfatiza donde D toma valores grandes [Thom85]. Ambas formas de control de malla pueden combinarse (ver sección V.3.3).

Las mallas no estructuradas se usan principalmente para la resolución mediante elementos finitos. Entre los métodos no estructurados, podemos encontrar entre otros:

- Triangulación de Delaunay [Dela34], [Laws77], [Wats81], [Hsua97]. Consiste en triangularizar un dominio de forma que:
  - Ningún vértice de ningún triángulo esté contenido en la circunferencia que pasa por los 3 vértices de cada triángulo de la malla.
  - La triangulación de Delaunay maximiza el ángulo mínimo de los elementos.

Normalmente, a la hora de triangularizar un dominio se sigue un procedimiento incremental (figura V.6), se comienza dividiendo el contorno, se comprueba si la solución obtenida cumple una serie de restricciones (por ejemplo, que el área de los triángulos, cada uno de los ángulos, o la longitud de cada uno de los lados sea menor que una cierta cantidad, etc.), si no se cumple se inserta un nodo en el interior del dominio (normalmente en el circuncentro del triángulo que no cumple la restricción), y se vuelve a triangularizar (en un entorno del punto insertado solamente), repitiéndose

este proceso hasta que se cumplen las restricciones. Este algoritmo se conoce como Bowyer-Watson [Bowy81] [Wats81].

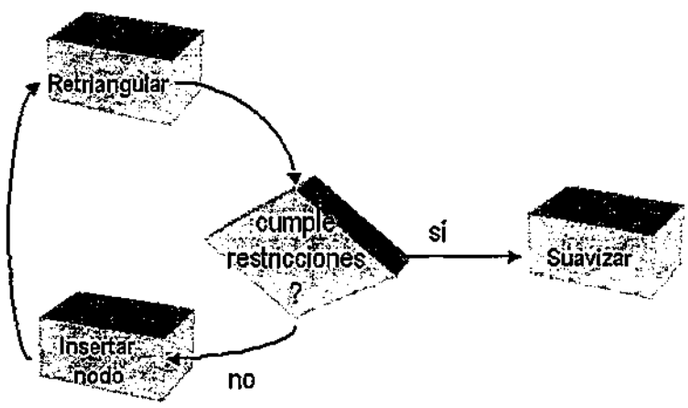


Figura V.6: Triangulación incremental.

Otra forma de triangulación, conocida como triangulación colectiva (figura V.7) eligen la posición inicial de todos los nodos en un primer paso, triangularizan, y opcionalmente aplican un suavizado a la malla. La principal dificultad de estos métodos es que una mala elección de la posición inicial de los nodos puede hacer que la malla final sea mala.

Figura V.7: Triangulación colectiva.

- Métodos basados en rejillas [Baeh87]. En estos métodos, una malla rectangular o triangular se superpone al dominio, recortándose la malla en los bordes para que se adapte al dominio. Este método es rápido, pero las mallas resultantes tienen poca calidad en los bordes, que es donde se requieren mallas con alta calidad.

- Avance frontal [Pera87]. Este método se basa en una idea sencilla:

  - Discretizar el contorno. Este es el 'frente' inicial.

  - Añadir triángulos con al menos un lado en el 'frente'. Este paso actualiza el frente en cada iteración. El proceso termina cuando el frente queda vacío.

Aunque la idea es sencilla, los algoritmos para este tipo de generación suelen ser bastante complejos. También pertenece a los métodos de triangulación incremental.

V.1.2.1 Suavizado

Las mallas generadas de forma no estructurada, normalmente se mejoran mediante un proceso de suavizado [Fiel88]. Este proceso consiste en ajustar la posición de los nodos generados para obtener una malla más suave. El método más usual es el suavizado Laplaciano, que consiste en resolver la ecuación de Laplace con la posición de los nodos.

$$X_i^{new} = \frac{1}{n_i} \sum_{j=1}^{n_i} X_j$$

Ecuación V.21: Suavizado Laplaciano.

La solución de esta ecuación mueve la posición de cada nodo hacia la media de las posiciones de sus vecinos.

Un nuevo enfoque emergente en la actualidad es el concepto de 'pliant mesh generation'. Esta filosofía consiste en agrupar el suavizado, la inserción y el borrado de nodos en cada iteración. La triangularización se puede hacer en esta iteración, o bien como un post-proceso (figuras V.8 y V.9).

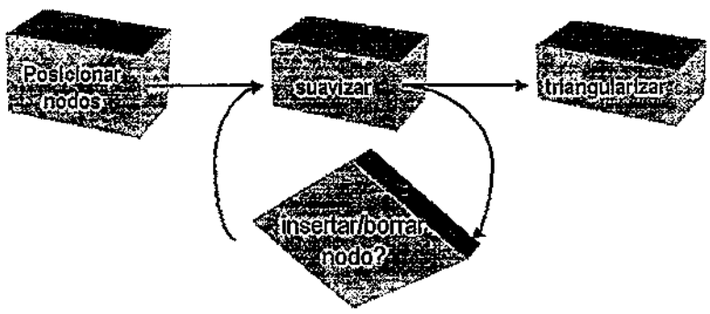


Figura V.8: 'Pliant mesh generation' con post-triangulación.

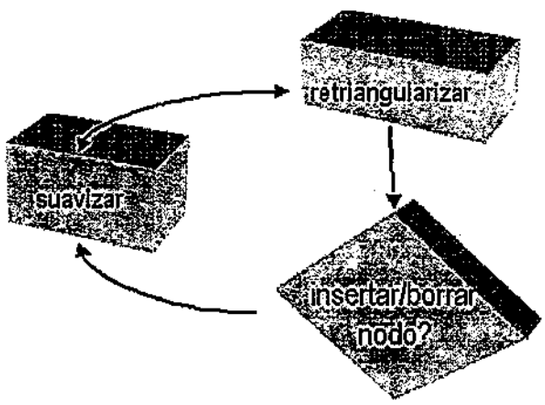


Figura V.9: 'Pliant mesh generation' con retriangulación.

■ V.1.3 Sistemas existentes de resolución de *PDEs*

Hay infinidad de sistemas para la resolución de *PDEs*, tanto comerciales como en fase de desarrollo. Algunos de los sistemas de resolución de *PDEs* comerciales más utilizados son:

- Abaqus [Abaq00]
- Cosmos [Stru00]
- NASTRAN [NE/NA00]

Aunque todos ellos superan ampliamente tanto en eficiencia como en potencia de cálculo a *OOCSMP*, ninguno de ellos satisfacía nuestras necesidades, ya que:

- No son herramientas de simulación propiamente dichas, su objetivo es la resolución de *PDEs* de la forma más eficiente.

- No son orientados a objetos.
- Siguen un esquema Preproceso-Proceso-Postproceso que no se adecua al paradigma de Simulación Visual Interactiva que sigue nuestro sistema, en el que es posible una interacción y cambio de los parámetros de simulación mientras ésta se está ejecutando.
- Ninguno de ellos ofrece la posibilidad de resolución de *PDEs* a través de Internet.

V.1.4 Resolución de *PDEs* con *OOCSMP*

Se ha incorporado a *OOCSMP* la capacidad de resolver un subconjunto de *PDEs* [deLa99], [Alfo99a]. En particular, el lenguaje es capaz de resolver *PDEs* y sistemas de *PDEs* de segundo orden, en una o dos dimensiones espaciales, así como de realizar un análisis transitorio (solución de ecuaciones que dependen del tiempo) de las mismas. Puede resolver ecuaciones de los tres tipos, hiperbólicas, parabólicas y elípticas. Las ecuaciones pueden ser *cuasilineales*, es decir que los coeficientes de las derivadas pueden ser funciones de la variable dependiente (*u* en la ecuación V.22), pero no pueden ser funciones de las primeras derivadas de *u*. Es decir, no resuelve ecuaciones no lineales.

La ecuación más compleja que *OOCSMP* puede resolver es la siguiente:

$$C_{tt}(u,x,y,t,...)\frac{\partial}{\partial t}\left(c_{tt}(u,x,y,t,...)\frac{\partial u}{\partial t}\right)+C_{t}(u,x,y,t,...)\frac{\partial u}{\partial t}+$$
$$+A_{x}(u,x,y,t,...)\frac{\partial}{\partial x}\left(a_{x}(u,x,y,t,...)\frac{\partial u}{\partial x}\right)+A_{y}(u,x,y,t,...)\frac{\partial}{\partial y}\left(a_{y}(u,x,y,t,...)\frac{\partial u}{\partial x}\right)$$
$$+A_{xy}(u,x,y,t,...)\frac{\partial}{\partial x}\left(a_{xy}(u,x,y,t,...)\frac{\partial u}{\partial y}\right)+A_{yx}(u,x,y,t,...)\frac{\partial}{\partial y}\left(a_{yx}(u,x,y,t,...)\frac{\partial u}{\partial x}\right)+...$$
$$+B_{x}(u,x,y,t,...)\frac{\partial u}{\partial x}+B_{y}(u,x,y,t,...)\frac{\partial u}{\partial y}+a_{0}(u,x,y,t,...)u+f(u,x,y,t,...)=0$$

Ecuación V.22: Ecuación *PDE* modelo.

Donde C_{tt}, c_{tt}, C_t, A_x, A_y, a_y, A_{xy}, a_{xy}, A_{yx}, a_{yx}, B_x, B_y, a_0 y f son expresiones *OOCSMP* que pueden depender de *x*, *y*, (las dos coordenadas espaciales), *t* (*TIME*), *u* o de cualquier otra variable del modelo. Para el caso de un sistema de ecuaciones, el modelo se muestra en la ecuación V.23.

Donde, como en el caso anterior, las expresiones que multiplican a las derivadas, pueden depender de *x*, *y*, *TIME*, u^1, u^2, etc.

$$
\begin{aligned}
&+ C_{tt}^{1}(\bullet)\frac{\partial}{\partial t}\left(c_{tt}^{1}(\bullet)\frac{\partial u^{1}}{\partial t}\right) + C_{tt}^{2}(\bullet)\left(c_{tt}^{2}(\bullet)\frac{\partial u^{2}}{\partial t}\right) + \ldots \\
&+ C_{t}^{1}(\bullet)\frac{\partial u^{1}}{\partial t} + C_{t}^{2}(\bullet)\frac{\partial u^{2}}{\partial t} + \ldots \\
&+ A_{x}^{1}(\bullet)\frac{\partial}{\partial x}\left(a_{x}^{1}(\bullet)\frac{\partial u^{1}}{\partial x}\right) + A_{x}^{2}(\bullet)\frac{\partial}{\partial x}\left(a_{x}^{2}(\bullet)\frac{\partial u^{2}}{\partial x}\right) + \\
&+ A_{y}^{1}(\bullet)\frac{\partial}{\partial y}\left(a_{y}^{1}(\bullet)\frac{\partial u^{1}}{\partial x}\right) + A_{y}^{1}(\bullet)\frac{\partial}{\partial y}\left(a_{y}^{2}(\bullet)\frac{\partial u^{2}}{\partial x}\right) \\
&+ A_{xy}^{1}(\bullet)\frac{\partial}{\partial x}\left(a_{xy}^{1}(\bullet)\frac{\partial u^{1}}{\partial y}\right) + A_{xy}^{2}(\bullet)\frac{\partial}{\partial x}\left(a_{xy}^{2}(\bullet)\frac{\partial u^{2}}{\partial y}\right) + \ldots \\
&+ A_{yx}^{1}(\bullet)\frac{\partial}{\partial y}\left(a_{yx}^{1}(\bullet)\frac{\partial u^{1}}{\partial x}\right) + A_{yx}^{2}(\bullet)\frac{\partial}{\partial y}\left(a_{yx}^{2}(\bullet)\frac{\partial u^{2}}{\partial x}\right) + \ldots \\
&+ B_{x}^{1}(\bullet)\frac{\partial u^{1}}{\partial x} + B_{x}^{2}(\bullet)\frac{\partial u^{2}}{\partial x} + \ldots + B_{y}^{1}(\bullet)\frac{\partial u^{1}}{\partial y} + B_{y}^{2}(\bullet)\frac{\partial u^{2}}{\partial y} + \ldots \\
&+ a_{0}^{1}(\bullet)u^{1} + a_{0}^{2}(\bullet)u^{2} + \ldots + f(\bullet) = 0
\end{aligned}
$$

Ecuación V.23: Sistema de *PDEs* modelo.

Las *PDEs* se pueden resolver en dominios uni o bidimensionales. En *OOCSMP* existe una serie de primitivas para la construcción de dominios. Los dominios básicos se discretizarán, asignándoles un generador de mallas y una ecuación/sistema de ecuaciones, así como un método de resolución. Es posible concatenar las mallas para conseguir mallas más complejas, de tal forma que para un mismo problema, se pueden haber generado pedazos de malla con métodos distintos, así como haber resuelto las ecuaciones con métodos también distintos.

Los métodos de generación de mallas que se han implementado en *OOCSMP* son:

- Métodos isoparamétricos.
- Triangulación de Delaunay.
- Generación Elíptica.

En cuanto a los métodos de resolución, se han implementado:

- Esquema explícito clásico de diferencias finitas [Lapi82] [Stri89].
- Esquema explícito de diferencias finitas de DuFort-Frankel [Lapi82] [Stri89].
- Esquema implícito de diferencias finitas, Crank-Nicolson [Cran47], para el caso de una dimensión, o *ADI* [Gour70] para el caso de dos dimensiones.
- Elementos Finitos [Otto92] [Redd93].

También es posible definir los dominios y las mallas dinámicamente, durante la simulación, mediante la herramienta *MGEN*. *MGEN* está basado en las mismas librerías Java que las primitivas de generación de mallas *OOCSMP*. Es un generador de mallas con fines principalmente educativos, si bien responde a algunas de las necesidades actuales de la generación de mallas [Thom99] como son:

- Fácil cambio de la geometría de la malla. Debido a que se separa la topología de la malla, basta con cambiar el tipo de generador de malla que se quiere usar. Además existen primitivas para la rotación, translación y escala de dominios y mallas. El uso de objetos, nos va a permitir encapsular la geometría dentro de clases *OOCSMP*, y declarar objetos en los

que se van a poder parametrizar varios aspectos de esta geometría, tales como la posición de los dominios básicos, los nodos de la malla, etc.

- Capacidades batch y gráfica. La funcionalidad de *MGEN* se puede reproducir al 100% con el lenguaje *OOCSMP*. Además *MGEN* también dispone de una ventana de mandatos, pudiendo así mismo leer código *OOCSMP*.

Los pasos básicos que hay que seguir para resolver una PDE, se muestran en la figura V.10. Estos son los pasos básicos para solucionar un problema sencillo, si bien para solucionar problemas más complejos, puede ser necesario, por ejemplo aplicar alguna operación geométrica a las mallas en el bucle de simulación (ver problema VII.3.2, que simula el calentamiento de dos piezas móviles) y resolver la *PDE* con la nueva configuración de la malla en cada pasada de este bucle. Estas operaciones geométricas también se pueden incluir en la parte declarativa del modelo, con lo que sólo se ejecutarían una vez, al comienzo de la simulación (ver por ejemplo el problema V.6.1, donde se realizan operaciones de translación sobre los dominios en el constructor de los objetos).

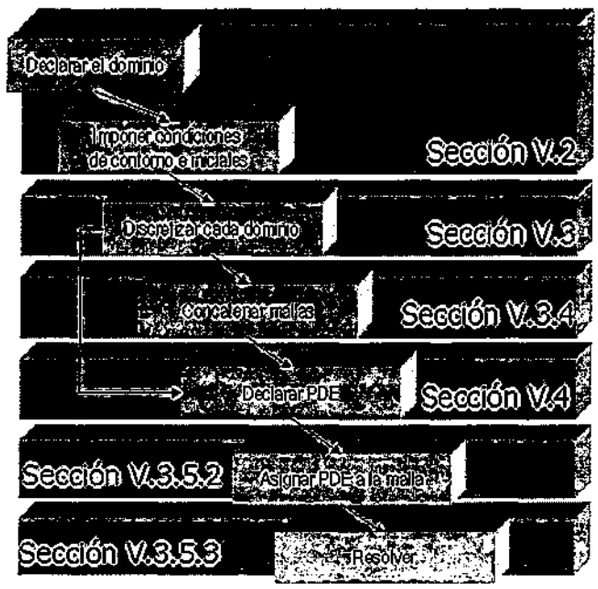


Figura V.10: Pasos para la resolución de una *PDE* con *OOCSMP*.

Además, se debe declarar una *PDE* por cada sub-malla, de tal forma que se ha de asignar un bloque *PDE* distinto (aunque represente la misma ecuación) a cada trozo de malla. El último paso 'Resolver' se puede incluir en cualquier sección *OOCSMP* (y se reordenará con el resto de ecuaciones), aunque si la ecuación no depende del tiempo, es conveniente ponerla en la sección, para que no se ejecute una vez por cada paso de tiempo. Todos los pasos, excepto el último se pueden efectuar bien con las primitivas *OOCSMP* que se detallan a continuación, o bien gráficamente con *MGEN*. Un ejemplo práctico que sigue este procedimiento, se presenta en el listado V.1 con la resolución de la ecuación del seno de Gordon.

Los aspectos novedosos que ha introducido la posibilidad de solucionar *PDEs* con *OOCSMP* son:

- Posibilidad de aprovechar la orientación a objetos cuando se soluciona una *PDE*.
- Resolución de *PDEs* a través de Internet, e implementación de una biblioteca de generación de mallas y de resolución de *PDEs* en Java.
- Es factible mezclar métodos de resolución de *PDEs*.
- Posibilidad de aprovechar las construcciones de un sistema de simulación continua, tales como el cálculo de cantidades auxiliares durante la simulación, modificación de las condiciones de contorno en el bucle de simulación, etc. Durante la simulación, se puede además manipular la geometría del problema, de tal forma que es posible rotar las mallas, transladarlas, etc.

■ V.2 Construcción de dominios

La declaración de dominios constituye el primer paso del procedimiento de la figura V.10. Los dominios en *OOCSMP* se declaran mediante primitivas, o bien se deja la definición para el momento de la ejecución de la simulación (con *MGEN*). La sintaxis de la declaración de estas primitivas es muy parecida, basta con indicar la posición de sus vértices (en sentido antihorario empezando por el inferior izquierdo), e imponer las condiciones de contorno e iniciales. La siguiente tabla muestra las primitivas para una y dos dimensiones, así como su aspecto.

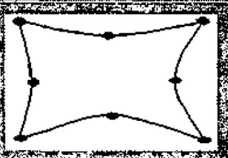
Dominio	Denominación OOCSMP	Aspecto
Cuadrilatero	QUADRILATERAL	
Triangulo	TRIANGLE	
Sector Circular	CIRCSECTOR	
Cuadrilatero de 8 nodos	QUAD8	
Linea	BAR	
Para el momento de la ejecución	MGEN	

Tabla V.1: Primitivas para la construcción de dominios.

Una vez discretizados estos dominios, es posible concatenarlos, para obtener así dominios más complejos.

Las condiciones de contorno (paso 2 de la figura V.10), esenciales, naturales o periódicas, también se especifican en la declaración de los dominios, y se pueden imponer sobre los lados, sobre los vértices o sobre nodos de la malla que será generada sobre el dominio. La sintaxis de cada uno de los dominios, se muestra en la siguiente tabla:

Dominio	Sintaxis
QUADRILATERAL	DOMAIN <id>:=QUADRILATERAL(<x1>,<y1>,<x2>,<y2>, <x3>,<y3>,<x4>,<y4>, <CONDICIONES>)
TRIANGLE	DOMAIN <id>:=TRIANGLE(<x1>,<y1>,<x2>,<y2>, <x3>,<y3>, <CONDICIONES>)
CIRCSECTOR	DOMAIN <id>:= CIRCSECTOR(<x>, <y>, <radExt>, <radInt>, <ang1>, <ang2> , <CONDICIONES>)
QUAD8	DOMAIN <id>:= QUAD8(<x1>,<y1>,<x2>,<y2>,<x3>,<y3>,<x4>,<y4>, <x5>,<y5>,<x6>,<y6>,<x7>,<y7>,<x8>,<y8>, <CONDICIONES>)
LINE	DOMAIN <id> := BAR (<x1>, <x2>, <CONDICIONES>
MGEN	DOMAIN <id>:=MGEN([DYNAMIC(<expresion>)] (,DYNAMIC(<expresion>))*)

Tabla V.2: Sintaxis de las primitivas para la construcción de dominios.

Donde *<CONDICIONES>* especifica las condiciones iniciales o de contorno. Las condiciones iniciales se imponen mediante *INITIAL*, o bien mediante *INITIALDT*, si estamos especificando condiciones iniciales derivativas. También se ha de indicar el grado de libertad para el que estamos definiendo la condición inicial, mediante *DOF( )*, si no se incluye, por defecto es 0. En el siguiente ejemplo se define un cuadrilátero, y se imponen condiciones iniciales para los dos primeros grados de libertad:

```
DOMAIN qd1 := QUADRILATERAL      (0,0,2,0,2,2,0,2,
                                 INITIAL(DOF(0),2*X*LOG(Y)),
                                 INITIAL(DOF(1),2*Y*LOG(X) )
```

Ejemplo V.1: Imposición de condiciones iniciales.

También se pueden especificar condiciones de contorno naturales, esenciales o periódicas. Mediante las construcciones *NATURAL*, *ESSENTIAL* o *PERIODIC*. Como en el caso de las condiciones iniciales, se ha de especificar el grado de libertad, y si se imponen sobre lados (*EDGE*), vértices (*CORNER*), o nodos de la malla (*NODE*). Los nodos y vértices se numeran en sentido antihorario empezando por el vértice/lado inferior izquierdo. En estas construcciones se pueden especificar listas de nodos/lados/vértices, separados mediante comas, así como intervalos cerrados. Los intervalos cerrados se definen por dos números (límite inferior y límite superior), separados por un *':'*. Los dos límites están incluidos en el intervalo. El siguiente ejemplo impone condiciones de contorno iniciales, esenciales y naturales sobre un triángulo:

```
DOMAIN td1 := TRIANGLE (0,0,4,0,2,3, INITIAL(2*(CH(X)+CH(Y)),
                       ESSENTIAL(EDGE(1,3),2*(SH(X)+SH(Y)),
                       NATURAL (CORNER(1:3),2*X*Y) )
```

Ejemplo V.2: Imposición de condiciones iniciales y de contorno.

Las condiciones de contorno periódicas sólo se pueden especificar sobre lados, estos normalmente serán opuestos. En este caso no hay que especificar ninguna expresión *OOCSMP*, ya que el valor de esta condición se obtiene mediante el método que se esté usando para calcular los valores de los nodos internos. Por ejemplo, para el caso de una línea:

```
DOMAIN l := BAR (0.0, 4.0,INITIAL(EXP(-X*X)), INITIALDT(0),
                          PERIODIC(EDGE(1,2)) )
```

Ejemplo V.3: Imposición de condiciones iniciales y periódicas.

Si en algún lado no se especifica ninguna condición, y el método de resolución necesita ese valor, aplicará condiciones de contorno numéricas (ver sección V.4.1)

En caso de que se quiera utilizar *MGEN* para el diseño de los dominios, la sintaxis es:

```
DOMAIN <id>:=MGEN([DYNAMIC(<exp>) [,DYNAMIC(<exp>)]*])
```

Sintaxis V.1: Uso de *MGEN* para diseñar un dominio.

Mediante la construcción *DYNAMIC(<exp>)*, indicamos que queremos usar la expresión *OOCSMP <exp>* como condición (de contorno o inicial) dentro de *MGEN*. La siguiente figura muestra el aspecto de *MGEN*, que será explicado con detalle en la sección V.7.

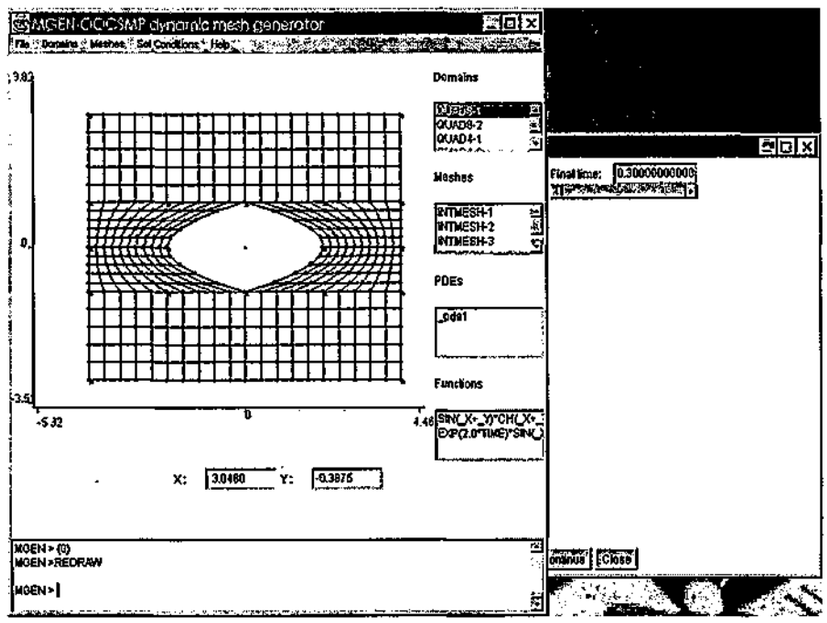


Figura V.11: Uso de *MGEN* durante una simulación.

V.2.1 Ejemplos de construcción de dominios

En esta sección, se va a dar un ejemplo de declaración de cada tipo de dominio.

Ejemplo 1. Sector Circular.

Un sector circular, con centro en (0,0), radio exterior 1.5 e interior 0.25, ángulo de inicio 0 y de final 90°, al que aplicamos una condición esencial (2*X) en el arco interno y una inicial (0) se definiría de la siguiente forma:

```
DOMAIN CS := CIRCSECTOR ( 0, 0, 1.5, 0.25, 0, 1.57,INITIAL(0),
                          ESSENTIAL(EDGE(1),2*X))
```

El resultado de dicha operación (con *MGEN*) tiene el siguiente aspecto:

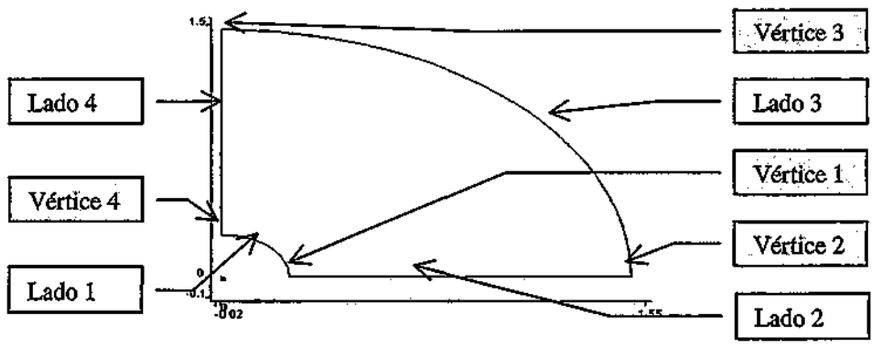


Figura V.12: Declaración de un dominio '*CIRCSECTOR*'

Ejemplo 2. Cuadrilátero.

Un cuadrilátero, con vértices en (-1.0, 1.0) , (1.0, 0.0) , (2.0, 2.0) y (-2.0, 2.0), al que aplicamos una condición natural *(2*X+TIME)* en todos los vértices y una inicial derivativa (0) se definiría de la siguiente forma:

```
DOMAIN Q41:=QUADRILATERAL(-1.0, 1.0, 1.0, 0.0, 2.0, 2.0,-2.0, 2.0,
                          NATURAL(CORNER(1:4),2*x*TIME),INITIALDT(0))
```

El resultado de dicha operación tendría el siguiente aspecto:

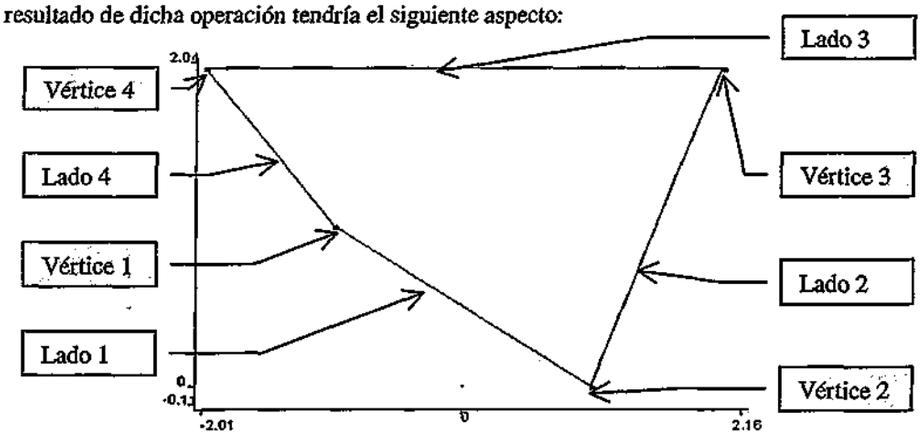


Figura V.13: Declaración de un dominio '*QUADRILATERAL*'

Ejemplo 3. Cuadrilátero de lados curvos.

Un *QUAD8* con coordenadas: (-1, -1), (0, -1.5), (1, -1), (1,0), (0.5,1), (0,0.75), (-0.5,1) y (-1,0), se declararía de la siguiente forma:

```
DOMAIN Q:=QUAD8(-1.0,-1.0, 0.0,-1.5, 1.0,-1.0, 1.0,
                0.0 0.50, 1.0 0.0, 0.75,-0.5, 1.0, -1.0, 0.0)
```

El resultado de dicha operación tendría el siguiente aspecto:

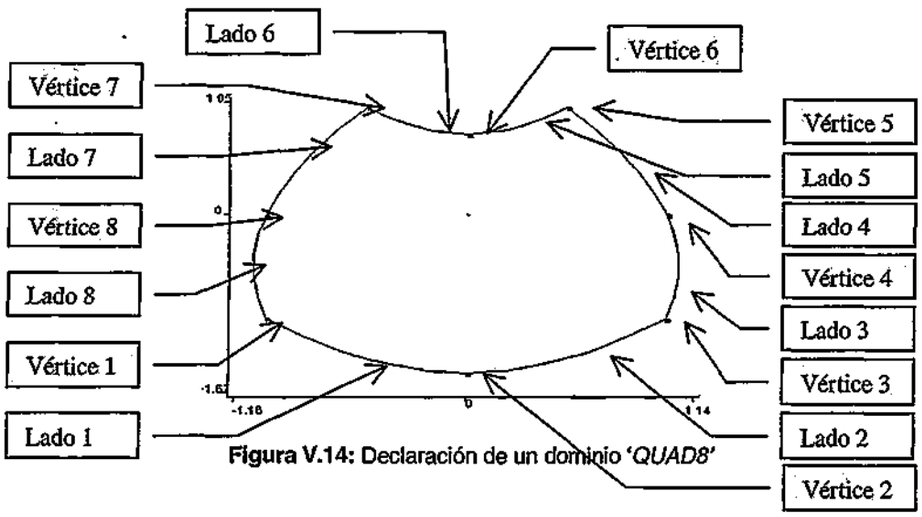


Figura V.14: Declaración de un dominio '*QUAD8*'

Ejemplo 4. Triángulo.

Un triángulo equilátero, de coordenadas: (-1, 0), (1, 0), (0,1.73205) se declararía de la siguiente forma:

```
DOMAIN T=TRIANGLE(-1.0,0.0, 1.0, 0,0, 0.0, 1.73205)
```

El resultado de dicha operación tendría el siguiente aspecto:

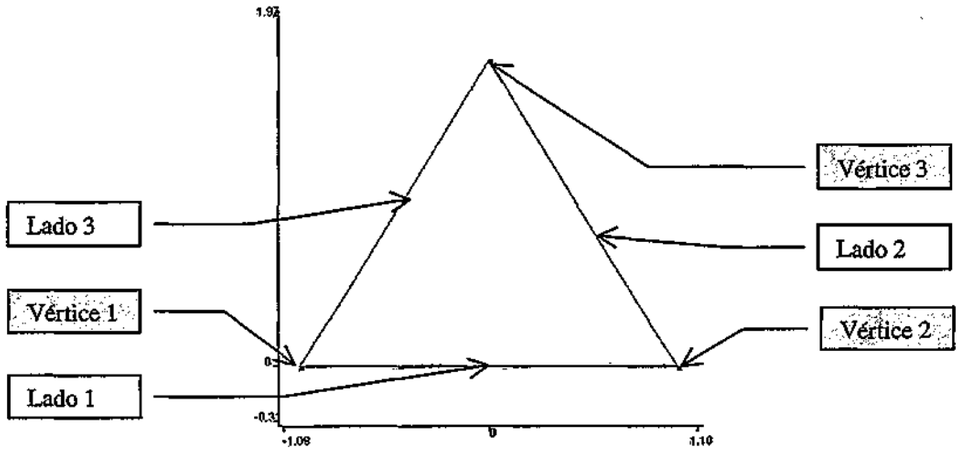


Figura V.15: Declaración de un dominio *'TRIANGLE'*

V.2.2 Operaciones sobre dominios

Una vez creados los dominios, es posible aplicarles las siguientes operaciones geométricas:

- Traslación (*MOVE*)
- Rotación (*ROTATE*) con respecto al circuncentro. Primero se realiza un cambio de coordenadas de todos los vértices del dominio al circuncentro, se le aplica la rotación a cada vértice, y se reestablecen las coordenadas. El ángulo de rotación ha de darse en radianes.
- Escala (*SCALE*) con respecto al circuncentro. Al igual que en el caso anterior, se realiza un cambio de coordenadas de todos los vértices del dominio al circuncentro, se le aplica la escala a cada vértice, y se reestablecen las coordenadas.

Se pueden invocar dichos métodos en la parte declarativa, antes de discretizarlos, o bien en la sección dinámica o en algún método o procedimiento. Estas operaciones no se propagan a la malla o a la *PDE* asociadas, en caso de que existan.

El ejemplo siguiente declara un cuadrilátero de 2x2 y esquina inferior izquierda en el origen. A continuación le aplica una translación de 2 unidades hacia la derecha, lo rota 45° y le aplica una magnificación del 50%.

```
DOMAIN qd1 := QUADRILATERAL(0, 0, 2, 0, 2, 2, 0, 2)
qd1.MOVE (2,0)
qd1.ROTATE(0.785375)
qd1.SCALE(1.5)
```

Ejemplo V.4: Aplicación de operaciones geométricas a los dominios en su declaración.

Posteriormente este dominio se puede discretizar. En el ejemplo, como las operaciones aparecen en la parte declarativa, se ejecutarán una vez, en la declaración del dominio. Si estas operaciones aparecen en la sección dinámica, se ejecutan una vez en cada pasada del bucle de simulación.

También se pueden invocar otros métodos sobre dominios para obtener la máxima coordenada de alguno de sus vértices (en las direcciones *X* e *Y*), o bien la mínima (en las direcciones *X* e *Y*). Los métodos no reciben ningún parámetro, y son los siguientes:

- *RIGHTX*
- *LEFTX*
- *TOPX*
- *BOTTOMX*
- *RIGHTY*
- *LEFTY*
- *TOPY*
- *BOTTOMY*

Estos métodos pueden aparecer en cualquier expresión *OOCSMP*, y al igual que el resto de los métodos sin parámetros de *OOCSMP*, se pueden invocar con o sin paréntesis.

V.3 Discretización de dominios: mallas

Una vez declarados los dominios, el siguiente paso es discretizarlos, esto constituye el paso tres del procedimiento de la figura V.10. Para ello, se ha de elegir un tipo de generador de malla, que 'rellena' el dominio mediante elementos básicos (símplices). En *OOCSMP* están disponibles los siguientes símplices:

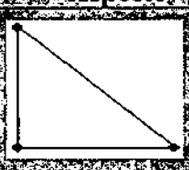
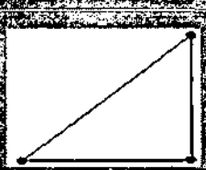
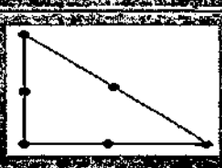
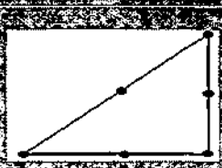
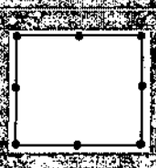
Tipo de símplice	Nombre OOCSMP	Aspecto
Triángulo de 3 nodos, orientado hacia la izquierda.	TRIANGLE3L	
Triángulo de 3 nodos, orientado hacia la derecha.	TRIANGLE3R	
Triángulo de 6 nodos, orientado hacia la izquierda.	TRIANGLE6L	
Triángulo de 6 nodos, orientado hacia la derecha.	TRIANGLE6R	
Cuadrilátero de 4 nodos.	QUADRILAT4	
Cuadrilátero de 8 nodos.	QUADRILAT8	
Línea de 2 nodos.	LINE2	
Línea de 3 nodos.	LINE3	

Tabla V.3: Símplices

Todos se pueden distorsionar para rellenar el mejor el dominio a discretizar. Aunque, a mayor distorsión, mayor error se va a cometer en la solución de la ecuación [Hugh87].

Las mallas se declaran de la siguiente forma:

```
MESH <id> := <tipo-generador> (
    <domain> ,  <element-type>
  [ , MAXSIZE(<max_size>)]
  [ , ELEMENTS(<number-x> [,<number-y>]) ]
  [ , SMOOTH ] )
```

Sintaxis V.2: Sintaxis general de declaración de una malla.

Donde :

- *<tipo-generador>*: Es el tipo de generador con el que se quiere generar la malla. Puede ser uno de los siguientes:
  - **DELAUNAY:** Triangulación mediante el algoritmo de Delaunay. Si *<element-type>* no es triangular, cada triángulo se divide en tres cuadriláteros, partiéndolos por su baricentro.
  - **ELLIPTIC:** Generador de mallas que resuelve dos ecuaciones elípticas en derivadas parciales. Una para la coordenada x y otra para la coordenada y [Thom85].
  - **ISOPARAMETRIC:** Generador de malla mediante elementos isoparamétricos [Hugh87].
- *<Domain>*: es el identificador del dominio a discretizar.
- *<Element-type>*: es uno de los 8 símplices anteriores.
- *MAXSIZE(<max_size>)*: Indica el máximo tamaño permitido para alguno de los lados del símplice.
- *ELEMENTS(<number-x> [, <number-y>])*: Número de elementos en X y en Y en el dominio transformado.
- *SMOOTH:* Indica si se quiere un suavizado Laplaciano de la malla generada.

A continuación, se describen en detalle cada uno de los generadores de mallas.

V.3.1 Triangulación de Delaunay

La triangulación de Delaunay, es un tipo de generación de mallas no estructurado. Este tipo de generador, recubre el dominio con triángulos, aunque en *OOCSMP*, también se puede recubrir mediante cuadrados, dividiendo cada triángulo generado en tres cuadrados. Utiliza el criterio de Delaunay, es decir, cada nodo de la triangulación no debe estar contenido dentro de la circunferencia que pasa por los tres vértices de cada triángulo.

La sintaxis es la siguiente:

```
MESH m := DELAUNAY (      <domain-id>, <simplice>,
                          ELEMENTS( <ex>, <ey>)
                          [, AREA (<max-area>) ]
                          [, ANGLE ( <min-angle>) ]
                          [, SIZE (<max-size>) ]
                          [, SMOOTH ] )
```

Sintaxis V.3: Triangulación de Delaunay.

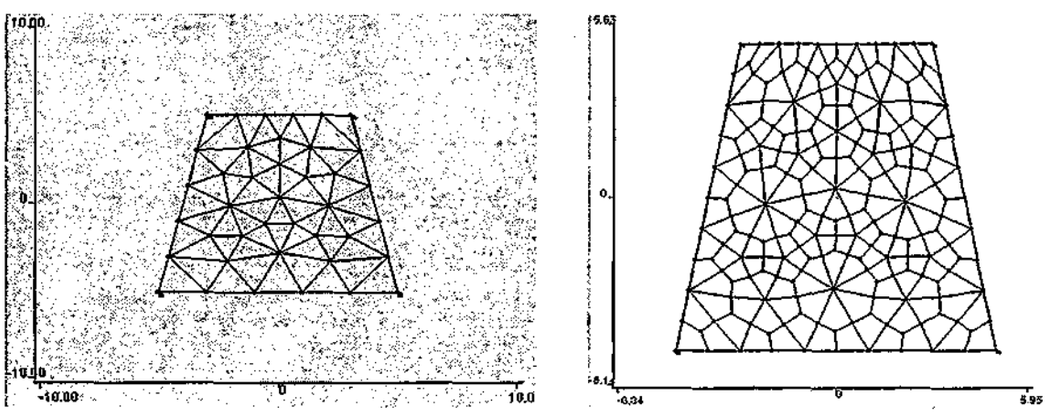
Donde :

- *<simplice>* indica el tipo de símplice que se usará para recubrir el dominio. Si se eligen cuadriláteros, todas las restricciones se aplicarán sobre los triángulos, y una vez generada la malla, se dividirá cada triángulo en tres cuadriláteros.
- *<max-area>* indica el máximo área que puede tener cada triángulo generado.
- *<min-angle>* indica el ángulo mínimo que puede tener algún ángulo de los triángulos generados.
- *<max-size>* indica la máxima longitud de uno de los lados del triángulo.

Se pueden indicar cero o una restricción de cada tipo. El algoritmo procede incrementalmente, generando primero una triangulación para los nodos del contorno, e insertando un punto en el circuncentro de cada triángulo que no cumpla con las restricciones impuestas [Chew89], [Rupp92].

El siguiente ejemplo muestra cómo discretizar un dominio mediante la triangulación de Delaunay, con suavizado.

```
DOMAIN Q1=QUADRILATERAL(  -5.0000,-5.0000, 5.0000,-5.0000, 3.0000, 5.0000,
                          -3.0000, 5.0000)
MESH D1=DELAUNAY(Q1, TRIANGLE3L, ELEMENTS(5,5), AREA( 2.0000), SMOOTH )
```



(a) (b)

Figura V.16a, b: Discretización de un cuadrilátero mediante Delaunay (triángulos (a) y cuadriláteros (b))

■ V.3.2 Generación Isoparamétrica.

Este generador está basado en la discretización por medio de elementos isoparamétricos, excepto en el caso del sector circular, que se discretiza mediante la transformación de coordenadas polares a cartesianas.

La sintaxis es la siguiente:

```
MESH <m-id> := ISOPARAMETRIC (<domain-id>, <simplice>,
                              ELEMENTS( <ex>, <ey>)
                              [, SMOOTH ] )
```

Sintaxis V.4: Generador isoparamétrico.

Con el mismo significado que en el caso anterior. Este tipo de generador también es capaz de generar triángulos, generando primero cuadriláteros y uniendo luego dos de sus vértices. Es de notar que en este caso, sí tiene sentido distinguir entre símplices triangulares orientados hacia la izquierda y hacia la derecha.

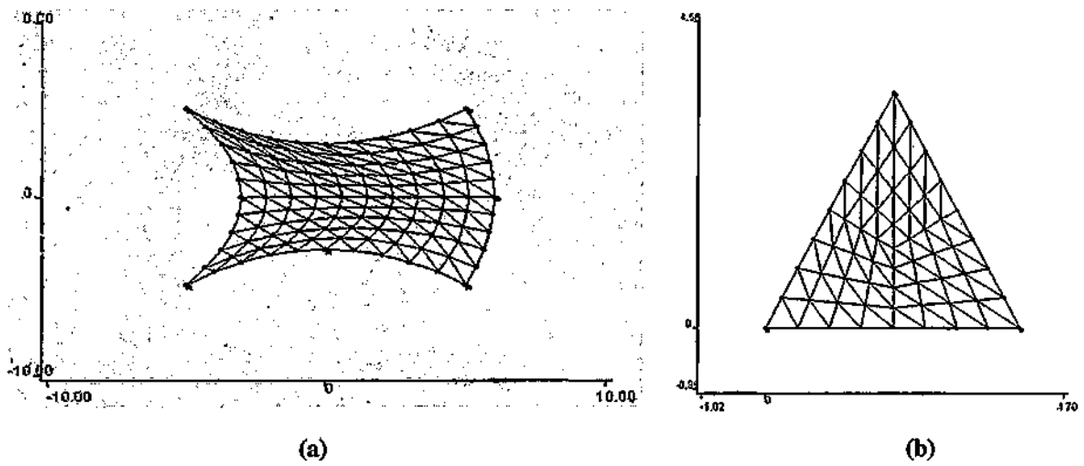
El siguiente ejemplo muestra cómo discretizar un cuadrilátero de nodos curvos, y un triángulo mediante este tipo de generador.

Cuadrilátero de Lados Curvos (a)

```
DOMAIN Q1=QUAD8(-5.0,-5.0, 0.0,-3.0,
       5.0,-5.0, 6.0, 0.0,
       5.0, 5.0, 0.0, 3.0,
       -5.0, 5.0,-3.0, 0.0)

MESH I1=ISOPARAMETRIC(Q1,TRIANGLE3L,
       ELEMENTS(10,10))
```

Triángulo (b)

```
TRIANGLE T1=TRIANGLE( 0,0, 4,0, 2,3.465)

MESH I2=ISOPARAMETRIC(T1,TRIANGLE3L,
       ELEMENTS(8,8))
```



(a) (b)

Figura V.17a, b: Discretización de un cuadrilátero de lados curvos (a) y de un triángulo (b) mediante elementos isoparamétricos.

Para discretizar un triángulo mediante este método, es necesario dividirlo primero en tres cuadriláteros, discretizar cada uno de ellos, y luego concatenarlos. Para este caso, el número de elementos en X y en Y no pueden ser ambos impares.

■ V.3.3 Generación Elíptica.

Como ya se comentó en el apartado V.1.2, este generador, discretiza el dominio resolviendo dos ecuaciones en derivadas parciales; una para la coordenada *x* y otra para la coordenada *y*. La ecuación que se resuelve por defecto es la ecuación de *Laplace*. Esta ecuación nos garantiza que el máximo y el mínimo de las dos coordenadas las vamos a encontrar en los bordes, ya que las ecuaciones elípticas siguen el principio del máximo y el mínimo. Esto es precisamente lo que queremos, que las coordenadas se distribuyan uniformemente en el interior del dominio, y que tanto el máximo como el mínimo estén en los bordes, de hecho, esta es la ecuación que usamos cuando aplicamos la operación de suavizado (*SMOOTH*).

La sintaxis para este tipo de generadores es la siguiente:

```
MESH <m-id> := ELLIPTIC (  <domain-id>, <simplex-id>,
                                ELEMENTS( <ex>, <ey>)
                           [, <sistema-pdes>] )
```

Sintaxis V.5: Generador Elíptico.

Donde `<sistema-pdes>` es un sistema de dos ecuaciones en derivadas parciales, definido previamente, que indica qué ecuación se debe resolver para cada coordenada (ver sección V.5 para la sintaxis de la declaración de los sistemas de ecuaciones). Basta con indicar una de las ecuaciones del sistema, el compilador añade el resto de las ecuaciones.

Por ejemplo, el siguiente modelo crea una malla elíptica sobre un rectángulo, con dos atractores para las líneas de coordenadas, uno en la esquina superior derecha, y otro simétrico en la esquina inferior izquierda. Para ello, en lugar de resolver el sistema de ecuaciones por defecto, vamos a resolver el siguiente sistema:

$$e^{(x+y)^2}\frac{\partial}{\partial\xi}\left(y\cdot dens(\xi,\mu)\frac{\partial x}{\partial\xi}\right)+e^{(x+y)^2}\frac{\partial}{\partial\mu}\left(y\cdot dens(\xi,\mu)\frac{\partial x}{\partial\mu}\right)+disc(\xi,\mu)\cdot dens(\xi,\mu)\cdot x+disc(\xi,\mu)\cdot dens(\xi,\mu)$$

$$e^{(x+y)^2}\frac{\partial}{\partial\xi}\left(x\cdot dens(\xi,\mu)\frac{\partial y}{\partial\xi}\right)+e^{(x+y)^2}\frac{\partial}{\partial\mu}\left(x\cdot dens(\xi,\mu)\frac{\partial y}{\partial\mu}\right)+disc(\xi,\mu)\cdot dens(\xi,\mu)\cdot y+disc(\xi,\mu)\cdot dens(\xi,\mu)$$

Ecuaciones V.24 a, b: Sistema a resolver

Donde las funciones *disc* y *dens* se han definido como :

$$disc(x,y)=\begin{cases}(x+y)>0, e^{x+y}\\ otro, e^{-(x+y)}\end{cases}$$

$$dens(x,y)=\begin{cases}\sqrt{x^2+y^2}\le 1/2, 1/(x+y+2)\\ (x+y)>0, e^{x+y}\\ otro, e^{-(x+y)}\end{cases}$$

Ecuaciones V.25 a, b: Funciones usadas por el sistema

Básicamente la función *disc* discrimina si el punto se encuentra a la derecha o a la izquierda de la recta *y=-x*, devolviendo valores simétricos. Además, la función *dens* también comprueba si el punto se encuentra cerca del centro del dominio. El modelo *OOCSMP* para conseguir la malla anterior se muestra en el listado V.1, el resultado de la ejecución de dicho modelo se muestra en la figura V.18.

```
DISC X, Y
  INSW(X+Y, DISC:= EXP(-(X+Y)), DISC:= EXP(X+Y) )

DENS X, Y
  INSW(SQRT(X*X+Y*Y)-0.5, DENS:= 1/(X+Y+2), DENS:=DISC(X,Y) )

DOMAIN qd := QUADRILATERAL(-1, -1, 1, -1, 1, 1, -1, 1)

PDE DEX (System(DEX,DEY)
             ,0, 0, 0, 0, 0, 0
             ,EXP((X+Y)*(X+Y)), DEY*DENS(X,Y), 0, 0
             ,EXP((X+Y)*(X+Y)), DEY*DENS(X,Y), 0, 0
             ,0, 0, 0, 0
             ,0, 0, 0, 0
             ,0, 0, 0, 0
             , DENS(X,Y)*DISC(X,Y), 0, DENS(X,Y)*DISC(X,Y), EXPLICIT)

PDE DEY (System(DEX,DEY)
             ,0, 0, 0, 0, 0, 0
             ,0, 0, EXP((X+Y)*(X+Y)), DEX*DENS(X,Y)
             ,0, 0, EXP((X+Y)*(X+Y)), DEX*DENS(X,Y)
             ,0, 0, 0, 0
             ,0, 0, 0, 0
             ,0, 0, 0, 0
             ,0, DENS(X,Y)*DISC(X,Y), DENS(X,Y)*DISC(X,Y), EXPLICIT)
MESH   m := ELLIPTIC( qd, QUADRILAT4, ELEMENTS(20,20), DEX )
TIMER FINTIM := 1.0, delta := 1.0
GRIDPLOT m
```

Listado V.1: Definición de una malla elíptica

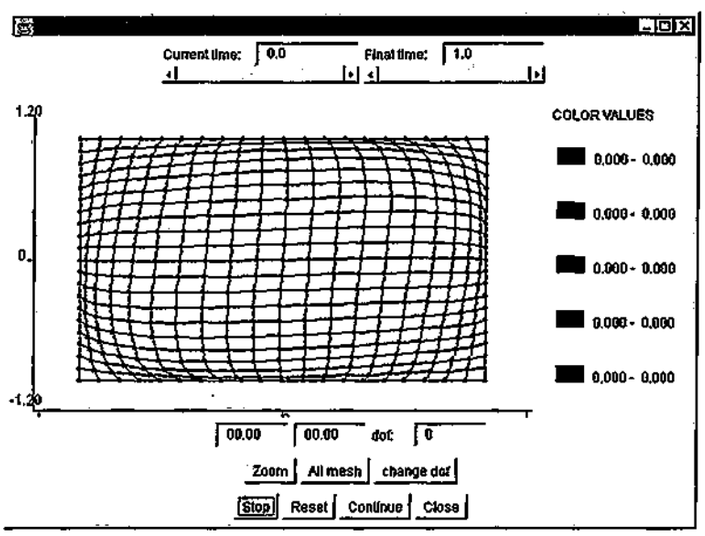


Figura V.18: Malla elíptica definida por el sistema de ecuaciones V.24

■ V.3.4 Concatenación de mallas.

Esta operación constituye el cuarto paso del procedimiento de la figura V.10. Dos mallas que tengan algún borde con algún punto en común se pueden concatenar para formar una malla más compleja. Esto constituye el paso cuatro del procedimiento de la figura V.10. La sintaxis es la siguiente:

```
<id-Malla-1>.CONCAT (<id-Malla-2> (,<id-Malla-n>)*)
```

Sintaxis V.6: Concatenación de mallas.

Donde :

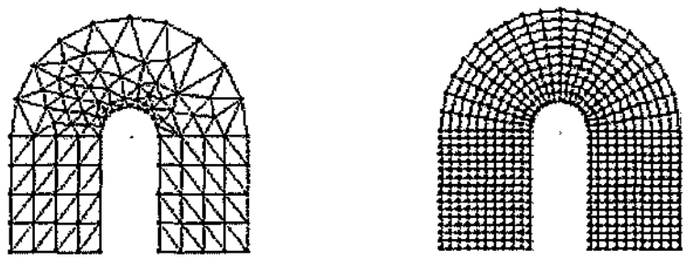
- `<id-Malla-1>` es el identificador de la malla a la que se concatena el resto.
- `<id-Malla-2>,..., <id-Malla-n>` son los identificadores del resto de las mallas que se concatenan.

El resultado de esta operación es que la malla 1 pasa a ser una malla compuesta, formada por la concatenación de todas las mallas. Idem con `<id-Malla-2>,..,<id-Malla-n>` . Es decir, la sintaxis anterior es equivalente a la siguiente:

```
<id-Malla-2>.CONCAT (<id-Malla-1> (,<id-Malla-n>)*)
```

Sintaxis V.7: Sintaxis equivalente a la Sintaxis V.6.

Por ejemplo, la siguiente figura es el resultado de la concatenación de un arco y dos cuadriláteros, discretizados mediante una malla interpolatoria.



(a) (b)

Figura V.19a,b: Concatenación de un arco y dos cuadriláteros (a) mezclando triangulación de Delaunay y generación de tipo interpolatoria (b) sólo con generación interpolatoria

Al concatenar dos mallas, las condiciones de contorno que queden en el interior de la malla se pierden. Esto nos va a permitir cambiar la geometría del problema, aplicando a las mallas operaciones geométricas (ver apartado V.3.5) y luego concatenándolas, las condiciones de contorno se ajustan automáticamente a la nueva geometría (ver ejemplo V.6.1).

Se pueden concatenar mallas bien en la parte declarativa o bien dentro de algún bloque. Si cuando se concatenan dos mallas, ambas tenían asignadas una ecuación y el tipo de resolución es el mismo, entonces el *solver* de la malla que recibe el método *CONCAT* ajusta sus datos, de forma que para solucionar el problema sólo hay que invocar el método *STEP* sobre esta primera malla. Esta es una operación costosa en tiempo, ya que por ejemplo, en el caso de resolución mediante elementos finitos, el *solver* tiene que volver a dimensionar vectores, copiar la solución de las mallas argumentos en sus propios datos, calcular los jacobianos de todos los elementos, etc. Es más eficiente concatenar mallas antes de haberles asignado una ecuación, si bien, a veces es inevitable concatenarlas en la parte no declarativa de la simulación.

V.3.5 Otras operaciones sobre mallas.

Al igual que a los dominios, a las mallas también se les pueden aplicar las operaciones geométricas de traslación, rotación y escala, con igual sintaxis. Al aplicar cualquiera de las operaciones anteriores a una malla, se aplica también al dominio asociado. Al igual que con los dominios, se puede especificar la operación, bien en la parte declarativa del modelo, o bien en la sección dinámica o en algún método o procedimiento. Si la malla es compuesta, la operación se aplica a cada sub-malla, teniendo en cuenta que el circuncentro, necesario en las operaciones de rotación y escala, se obtiene entonces considerando todos los vértices de las sub-mallas.

Los métodos para obtener la máxima coordenada de alguno de sus vértices (en las direcciones *X* e *Y*), o bien la mínima (en las direcciones *X* e *Y*) también se pueden aplicar a las mallas, con igual sintaxis que para los dominios.

Las mallas también admiten las siguientes operaciones:

V.3.5.1 Separación de mallas

Dos mallas concatenadas se pueden separar invocando el método *DETACH* a la malla a la que se aplicó el método *CONCAT*. *DETACH* lleva un solo parámetro que es la malla de la que se quiere separar la primera malla. La configuración de las condiciones de contorno queda como antes de concatenar las mallas. Si ambas mallas tienen asignada una ecuación, los *solvers* han de ajustar de nuevo sus datos, y para solucionar ahora las ecuaciones, se ha de invocar el método *STEP* sobre cada una de las mallas.

V.3.5.2 Asignación de una *PDE* para que sea resuelta en la malla

Esta operación constituye el sexto paso del procedimiento de la figura V.10. La *PDE*, o el sistema de *PDEs* se asignan a la malla mediante el método:

```
<malla>.setPDE( <pde-1> (, <pde-n> )* )
```

Sintaxis V.8: Asignación de una *PDE* a una malla, en la que será resuelta.

Si la malla es compuesta, se ha de asignar una *PDE* a cada submalla. De esta forma, sería posible resolver ecuaciones distintas en diversas zonas de la malla, o la misma ecuación, pero con distinto método de resolución. Esta posibilidad será vista en detalle en la sección V.4.7.

Si se quiere resolver un sistema de *PDEs*, se han de indicar todas las ecuaciones que componen el sistema dentro de este método.

V.3.5.3 Resolución de la *PDE*

Esta operación constituye el séptimo y último paso del procedimiento de la figura V.10. Una vez asignada la *PDE* a la malla, para resolver, sólo hay que invocar el método *STEP()* sobre la malla. Si la malla es compuesta, hay que invocar el método sobre la malla principal (aquella sobre la que se invocó el método *CONCAT*). Si la ecuación es dependiente del tiempo, hay que invocar el método en cada pasada del bucle de simulación. Si no lo es, se puede invocar en la sección initial, o si se invoca en la sección *DYNAMIC*, se resolverá en cada pasada del bucle de simulación.

V.3.5.4 Obtención de los valores de la *PDE*

Se puede obtener el valor de la *PDE* en un punto de la malla mediante el método *VALUE*. Este método puede aparecer en cualquier expresión *OOCSMP*. El método está sobrecargado, de forma que:

- Si se está resolviendo un sistema de ecuaciones en la malla, se puede especificar la malla de la que se quiere la solución, incluyendo la construcción *DOF(<grado-de-libertad>)*. Si no se especifica, por defecto es el cero.

- Si la malla es unidimensional y *VALUE* se aplica sólo con el parámetro *DOF*, o bien sin él, devuelve un vector con los valores de la ecuación en todos los puntos (ver sección VI.2.3). La sintaxis en este caso sería:

```
<malla>.VALUE([DOF(<gdl>)]) ó
<malla>.VALUE[()]
```

Sintaxis V.9: Sintaxis para mallas unidimensionales, obtención de todos los valores.

- Si la malla es unidimensional y se aplica con un parámetro (además de *DOF*), se entiende que el parámetro representa un punto de la malla, si no coincide con ningún nodo de la malla, se devuelve el valor resultante de aplicar interpolación lineal entre el valor anterior y el posterior. La sintaxis sería la siguiente:

```
<malla>.VALUE([DOF(<gdl>),] <coord-X>)
```

Sintaxis V.10: Sintaxis para mallas unidimensionales, obtención del valor en un punto.

- Si la malla es bidimensional, se ha de invocar el método con dos parámetros, que indican la coordenada de un punto de la malla. Si no coincide con ningún nodo de la malla, se devuelve el valor resultante de aplicar interpolación lineal o cuadrática (depende de con qué elementos estemos resolviendo la ecuación, ver sección V.3, los elementos con 3 nodos por lado dan interpolación cuadrática, con dos nodos por lado, dan interpolación lineal) del elemento en el que esté incluido el punto. La sintaxis sería pues:

```
<malla>.VALUE([DOF(<gdl>),] <coord-X>, <coord-Y>)
```

Sintaxis V.11: Sintaxis para mallas bidimensionales, obtención del valor en un punto.

V.3.5.5 Modificación de los valores de la PDE

Para modificar el valor de la ecuación en un punto de la malla, se ha de utilizar el método *setVALUE*, con las siguientes sintaxis:

```
<malla>.setVALUE([DOF(<gdl>),] <coord-X>, <expresion>)
```

Sintaxis V.12: Sintaxis para mallas bidimensionales, modificación del valor en un punto.

La sintaxis anterior se utiliza cuando la malla es unidimensional (ya que sólo se especifica la coordenada *x*). Si se está resolviendo un sistema de ecuaciones se utiliza *DOF* para indicar de qué ecuación se quiere modificar el valor. Si no se especifica, el valor de *DOF* es cero. El parámetro `<expresion>` puede ser cualquier expresión *OOCSMP*.

Para mallas bidimensionales, se utiliza la sintaxis:

```
<malla>.setVALUE([DOF(<gdl>),] <coord-X>, <coord-Y>, <expresion>)
```

Sintaxis V.13: Sintaxis para mallas bidimensionales, modificación del valor en un punto.

Con el mismo significado de los parámetros que para el caso anterior.

V.3.5.6 Añadir o cambiar las condiciones de contorno

Si bien lo normal es poner las condiciones de contorno cuando se declaran los dominios, también es posible ponerlas después de haberlos discretizado. Si se especifica una condición de contorno en un punto en el que ya había alguna especificada, está se sobreescribe con el nuevo valor. La sintaxis de estas operaciones es muy parecida a la usada para imponer las condiciones al declarar los dominios:

```
<malla>.ESSENTIAL([DOF(<gdl>),] <lugar>, <expresion-OOCSMP>)
<malla>.NATURAL([DOF(<gdl>),] <lugar>, <expresion-OOCSMP>)
```

Sintaxis V.14: Sintaxis para añadir o modificar condiciones de contorno.

Donde :

- El primer parámetro (*DOF*), es opcional y se refiere a la ecuación dentro del sistema de ecuaciones a la que imponemos la condición de contorno.
- *<lugar>* puede tomar los mismo valores que al establecer las condiciones en la declaración de los dominios, es decir: *NODE*, *EDGE* o *CORNER*, y seguido y entre paréntesis una lista o un intervalo de los elementos.
- *<expresion-OOCSMP>* puede ser cualquier expresión *OOCSMP*, incluida una función dependiente de las coordenadas *X* e *Y*.

Estas operaciones pueden aparecer bien en la parte declarativa del programa principal o de alguna clase, con lo cual la operación se realiza una vez, bien antes de iniciarse la simulación o en el constructor de la clase, o pueden llamarse en algún punto del bucle principal de simulación, esto implica que podemos cambiar el valor y el tipo de las condiciones de contorno durante la simulación.

■ V.4 Declaración de *PDEs*

La declaración de la ecuación a resolver constituye el paso número cinco del procedimiento de la figura V.10. Las *PDEs* se declaran genéricamente de la siguiente forma:

```
PDE <id> (<Ctt>,<Cxx>,<Ct>,
          <Ax>,<ax>,<Ay>,<ay>,<Axy>,<axy>,<Ayx>,<ayx>,
          <Bx>,<By>,<a0>,<f>,
          <PDE-METHOD>)
```

Sintaxis V.15: Sintaxis general para la declaración de una *PDE*.

Donde:

- Los 15 primeros parámetros son expresiones que representan las 15 funciones de la ecuación modelo (ecuación V.22).
- *<PDE-METHOD>* es el método de resolución para la *PDE*, que puede ser:
  - *EXPLICIT* : El método de diferencias finitas explícito clásico.
  - *DUFORTE* : El esquema de diferencias finitas explícito de DuForte-Frankel.
  - *IMPLICIT* : El método de Crank-Nicolson o *ADI* implícito.
  - *FEM* : El método de los elementos Finitos.

A continuación se explican las particularidades de cada método.

■ V.4.1 El método de las diferencias finitas explícito: *EXPLICIT*

Este método resuelve la(s) correspondiente(s) *PDE(s)* mediante el método de diferencias finitas clásico, incluso en dominios no rectangulares. Es posible parametrizar las aproximaciones de las derivadas en *X*, *Y* y en el tiempo, dando como resultados esquemas hacia delante (*forwards*), hacia atrás (*backwards*) o centradas en el tiempo y/o espacio (*X* y/o *Y*). Si no se especifica ninguno, el compilador elige el más adecuado teniendo en cuenta la geometría, la ecuación a resolver y las condiciones del problema. La sintaxis es la siguiente:

```
PDE <id> (<Ctt>,<ctt>,<Ct>,
          <Ax>,<ax>,<Ay>,<ay>,<Axy>,<axy>,<Ayx>,<ayx>,
          <Bx>,<By>,<a0>,<f>,
          EXPLICIT
          [,TScheme(<tipo-esquema>)]
          [,XScheme(<tipo-esquema>)]
          [,YScheme(<tipo-esquema>)]
          [,ITERATIVE(<method>)] )
```

Sintaxis V.16: Declaración de una *PDE*, resolución mediante un método explícito.

donde `<tipo-esquema>` puede ser *CENTRAL*, *FORWARD* o *BACKWARDS*, si bien las librerías de resolución pueden cambiar el tipo de esquema a un esquema correcto, dependiendo de las condiciones de contorno. El tipo de esquema por defecto para el tiempo y el espacio es el *CENTRAL*. La opción *ITERATIVE* se explica en el apartado V.4.3.

Así pues, las sustituciones de las derivadas que se realizan son, para las primeras derivadas:

Term.	Esquema Central	Esquema hacia adelante	Esquema hacia atrás	Errores de truncamiento
$\frac{\partial u}{\partial x}$	$\frac{U(r+1,s,t)-U(r-1,s,t)}{2h}$	$\frac{U(r+1,s,t)-U(r,s,t)}{h}$	$\frac{U(r,s,t)-U(r-1,s,t)}{h}$	$O(h^2),O(h),O(h)$
$\frac{\partial u}{\partial y}$	$\frac{U(r,s+1,t)-U(r,s-1,t)}{2k}$	$\frac{U(r,s+1,t)-U(r,s,t)}{k}$	$\frac{U(r,s,t)-U(r,s-1,t)}{k}$	$O(k^2),O(k),O(k)$
$\frac{\partial u}{\partial t}$	$\frac{U(r,s,t+1)-U(r,s,t-1)}{2\Delta}$	$\frac{U(r,s,t+1)-U(r,s,t)}{\Delta}$	$\frac{U(r,s,t)-U(r,s,t-1)}{\Delta}$	$O(\Delta^2),O(\Delta),O(\Delta)$

Tabla V.4: Discretizaciones de las derivadas de primer orden.

Donde *h*, *k* y *Δ*, son las distancias entre dos puntos de la malla en las direcciones *x*, *y* y el tiempo, respectivamente. *U(r,s,t)* representa el valor de la función *u* en el punto *(r*h, s*k)* en el instante *t*Δ*. Para el resto de términos, las sustituciones son las siguientes:

$$A_x(x,y,t,...)\frac{d}{dx}\left(a_x(x,y,t,...)\frac{dZ}{dx}\right)=A_x(x,y,t,...)\frac{a_x(x+h,y,t,...)\frac{dZ}{dx}-a_x(x,y,t,...)\frac{dZ}{dx}}{h}=$$

$$A_x(x,y,t,...)\frac{a_x(x+h,y,t,...)\big(Z(x+h,y,t...)-Z(x,y,t...)\big)-a_x(x,y,t,...)\big(Z(x,y,t...)-Z(x-h,y,t...)\big)}{h^2}$$

(a)

$$A_y(x,y,t,...)\frac{d}{dy}\left(a_y(x,y,t,...)\frac{dZ}{dy}\right)=A_y(x,y,t,...)\frac{a_y(x,y+k,t,...)\frac{dZ}{dy}-a_x(x,y,t,...)\frac{dZ}{dy}}{k}=$$
$$A_y(x,y,t,...)\frac{a_y(x,y+k,t,...)\big(Z(x,y+k,t...)-Z(x,y,t...)\big)-a_y(x,y,t,...)\big(Z(x,y,t...)-Z(x,y-k,t...)\big)}{k^2}$$

(b)

$$A_{xy}(x,y,t,...)\frac{d}{dx}\left(a_{xy}(x,y,t,...)\frac{dZ}{dy}\right)=A_{xy}(x,y,t,...)\frac{a_{xy}(x+h,y,t,...)\frac{dZ}{dy}-a_{xy}(x-h,y,t,...)\frac{dZ}{dy}}{2h}=$$
$$A_{xy}(x,y,t,...)\frac{a_{xy}(x+h,y,t,...)\big(Z(x+h,y+k,t...)-Z(x+h,y-k,t...)\big)-a_{xy}(x-h,y,t,...)\big(Z(x-h,y+k,t...)-Z(x-h,y-k,t...)\big)}{4hk}$$

(c)

$$A_{yx}(x,y,t,...)\frac{d}{dy}\left(a_{yx}(x,y,t,...)\frac{dZ}{dx}\right)=A_{yx}(x,y,t,...)\frac{a_{yx}(x,y+k,t,...)\frac{dZ}{dx}-a_{yx}(x,y-k,t,...)\frac{dZ}{dx}}{2k}=$$
$$A_{yx}(x,y,t,...)\frac{a_{yx}(x,y+k,t,...)\big(Z(x+h,y+k,t...)-Z(x-h,y+k,t...)\big)-a_{yx}(x,y-k,t,...)\big(Z(x+h,y-k,t...)-Z(x-h,y-k,t...)\big)}{4hk}$$

(d)

Ecuaciones V.26a,b,c y d: Discretizaciones de las derivadas segundas de la ecuación modelo.

Por ejemplo, para el caso de la ecuación $u_t+u_{xx}=0$, quedaría la siguiente molécula computacional, para un esquema hacia delante en el tiempo:

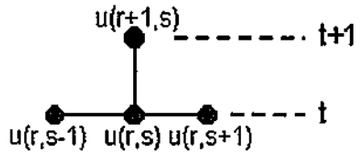


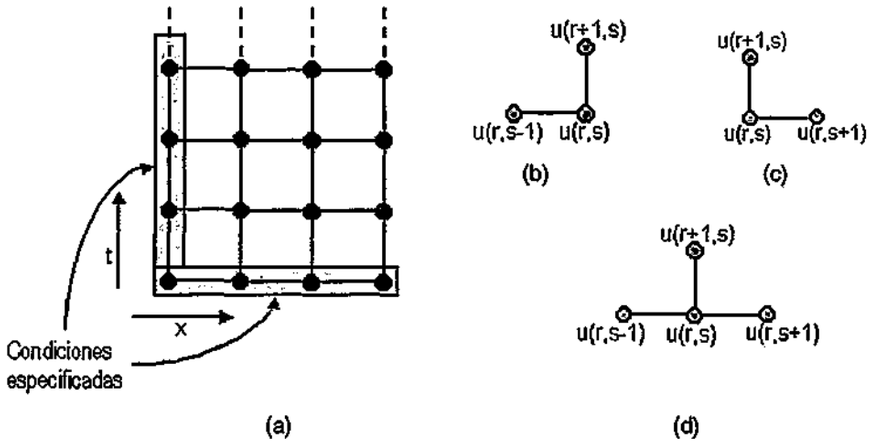
Figura V.20: Molécula computacional para la ecuación $u_t+u_{xx}=0$ y un esquema explícito

Como se puede observar, podemos calcular todos los valores de la fila *(t+1)* (es decir, del siguiente intervalo de tiempo) en función de los de la línea *t*, suponiendo que tenemos condiciones de contorno a la izquierda y a la derecha, es decir, el primer y el último nodo de cada fila ya están calculados, la molécula anterior permite calcular desde el segundo nodo hasta el penúltimo. Si no se tienen condiciones de contorno a la izquierda o a la derecha, el método aplica condiciones de contorno numéricas, que pueden ser de los siguientes tipos [Stri89]:

- Extrapolación de la solución de los puntos interiores de la malla, con expresiones del tipo: `u(t+1,s) = u(t+1,s-1)` o `u(t+1,s)=2u(t+1,s-1)-u(t+1,s-2)`.
- Extrapolación cuasi-característica, que son expresiones del tipo: `u(t+1,s)=u(t,s-1)` o `u(t+1,s)=2u(t,s-1)-u(t-1,s-2)` y que se llaman así porque la extrapolación se hace de puntos cerca de las características.

Para el caso de dos dimensiones, dependiendo de dónde se han puesto las condiciones de contorno, el método de resolución decide en qué dirección tiene que recorrer la malla, y qué esquema debe usar en la discretización espacial para conseguir calcular la solución en todos los puntos de la malla. Por ejemplo, para la ecuación de transporte en una dimensión ($u_t+au_x=0$) si $a>0$, tenemos que especificar una condición inicial y una de contorno en la parte izquierda del dominio, ya que la onda se desplaza de izquierda a derecha, por tanto el método ha de recorrer el dominio de izquierda a derecha, y en este caso, el método de resolución elegiría una discretización espacial hacia atrás, ya que la molécula computacional

que se obtiene nos permite calcular el último punto del dominio (ver figura V.21). Otros esquemas, dan lugar a moléculas que necesitan condiciones de contorno numéricas, lo que casi siempre deja más error.



Figuras V.21a,b,c,d: Geometría del problema, especificación de condiciones iniciales y de contorno (a). Moléculas computacionales para la ecuación $u_t+u_x=0$: backwards (b), forwards (c) y central (d)

Una posible mejora (no implementada), sería elegir un método centrado para calcular todos los nodos de una fila, excepto el último, y usar en este último un método backwards. Si bien esta mejora sólo es posible si se tienen sólo primeras derivadas, ya que una segunda derivada da lugar a una molécula del estilo de la figura V.21(d).

En la figura V.21(a) se observa que la condición inicial u de contorno se solapan en el nodo de la esquina inferior izquierda. Si bien, normalmente estas condiciones tienen el mismo valor en dicho nodo, en *OOCSMP*, se imponen primero las condiciones iniciales, y luego las de contorno (que sobreescribirían a las primeras en el contorno, en $t=0$, si no tuviesen el mismo valor en dichos puntos).

V.4.2 Resolución mediante diferencias finitas en dominios no cuadrados

La resolución de las ecuaciones en dominios no cuadrados, se realiza transformando cada porción básica del dominio al cuadrado de lado 2 y esquina inferior izquierda en (-1,-1), así mismo también se han de transformar las ecuaciones a resolver. En caso de que el dominio físico sea rectangular, el compilador lo detecta y no se realizan las transformaciones, ya que de esta forma el cálculo es mucho más rápido.

Es decir, las bibliotecas que se han construido, realizan una transformación del espacio *(x,y)* al *(ξ,η)*, donde el dominio de las variables ξ y η es el [-1..1]. Para ello, se conocen las funciones de transformación de cada primitiva básica de dominio, y sus derivadas hasta el orden dos. Las funciones de transformación son:

- x(ξ,η), a partir de ξ y η, calcula la coordenada x.
- y(ξ,η), a partir de ξ y η, calcula la coordenada y.
- ξ(x,y), a partir de x e y, calcula la coordenada ξ.
- η(x,y) , a partir de x e y, calcula la coordenada η.

Las transformaciones que se hacen de cada término de la ecuación original son las siguientes:

Término	Transformación
Función escalar A(x,y)	$A(x(\xi,\eta), y(\xi,\eta))$
$\frac{\partial u}{\partial x}$	$\frac{\partial u}{\partial \varepsilon}\frac{\partial \varepsilon}{\partial x}+\frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial x}$
$\frac{\partial u}{\partial y}$	$\frac{\partial u}{\partial \varepsilon}\frac{\partial \varepsilon}{\partial y}+\frac{\partial u}{\partial \eta}\frac{\partial \eta}{\partial y}$
$\frac{\partial^2 u}{\partial x^2}$	$\frac{\partial^2 u}{\partial \varepsilon^2}\frac{\partial \varepsilon}{\partial x}\frac{\partial \varepsilon}{\partial x}+2\frac{\partial u}{\partial \varepsilon \partial \eta}\frac{\partial \eta}{\partial x}\frac{\partial \varepsilon}{\partial x}+\frac{\partial^2 u}{\partial \eta^2}\frac{\partial \eta}{\partial x}\frac{\partial \eta}{\partial x}+\frac{\partial u}{\partial \varepsilon}\frac{\partial^2 \varepsilon}{\partial x^2}+\frac{\partial u}{\partial \eta}\frac{\partial^2 \eta}{\partial x^2}$
$\frac{\partial^2 u}{\partial y^2}$	$\frac{\partial^2 u}{\partial \varepsilon^2}\frac{\partial \varepsilon}{\partial y}\frac{\partial \varepsilon}{\partial y}+2\frac{\partial u}{\partial \varepsilon \partial \eta}\frac{\partial \eta}{\partial y}\frac{\partial \varepsilon}{\partial y}+\frac{\partial^2 u}{\partial \eta^2}\frac{\partial \eta}{\partial y}\frac{\partial \eta}{\partial y}+\frac{\partial u}{\partial \varepsilon}\frac{\partial^2 \varepsilon}{\partial y^2}+\frac{\partial u}{\partial \eta}\frac{\partial^2 \eta}{\partial y^2}$

Tabla V.5: Sustituciones de las derivadas en dominios no cuadrados.

y se resuelve la ecuación resultante en el dominio (ξ, η).

Como se mencionó anteriormente, x(ξ, η) e y(ξ, η) se han obtenido de la formulación isoparamétrica, excepto en el caso del sector circular, que se han obtenido del paso a coordenadas polares.

V.4.3 Solución de problemas elípticos

Si estamos resolviendo un problema elíptico (como por ejemplo, la ecuación de Laplace, ecuación V.19) al sustituir las derivadas, nos queda una molécula computacional del tipo:

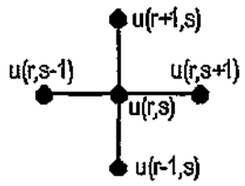


Figura V.22: *Molécula computacional para la ecuación de Laplace*

Donde los todos los nodos son incógnitas, a no ser que alguno de los 4 nodos exteriores pertenezcan al contorno, con lo que al final, debemos resolver un sistema de ecuaciones del tipo[Stou93]:

$$[A]\{u\}=\{b\}$$

Donde :

- *{u}* son las incógnitas, e.d. los nodos internos del dominio,
- *{b}* son los nodos del contorno, de los que conocemos la solución.

Si la malla es de de *(NxM)* nodos, resulta un sistema de *(N-2)*(M-2)* ecuaciones. Debido a que es un sistema de grandes dimensiones, conviene usar métodos aproximados (iterativos) de solución del sistema. Los métodos iterativos que se van a utilizar para resolver este tipo de problemas, descomponen la matriz *[A]* en expresiones del tipo *[A] =[D]+[L]+[U]*, y van calculando {u} en distintos pasos: $\{u\}^{n+1} = [G]\{u\}^n+\{c\}$. En *OOCSMP*, podemos elegir entre los siguientes:

- *GAUSS:* Resuelve mediante el método de Gauss-Seidel [Lapi82], realiza la siguiente sustitución:

$$[G]_{GS}=-([D]+[L])^{-1}[U],$$
$$\{c\}_{GS}=([D]+[L])^{-1}\{b\}$$

Ecuaciones V.27: *Sustitución del método de Gauss-Seidel.*

- *SOR:* Resuelve mediante el método de Seidel Over Relaxation [Lapi82]. Supongamos que tenemos la siguiente discretización y numeración de los nodos de un dominio:

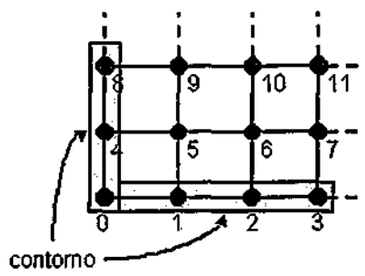


Figura V.23: Ejemplo de discretización.

Es decir, conocemos los valores de u_0, u_1, u_2, u_3, u_4 y u_8 debido a que están en el contorno. El método recorre los nodos por filas, calcula el valor de $u'_5{}^{n+1}$ mediante Gauss-Seidel y le aplica la expresión: $u_5^{n+1}=u_5^n+w(u'_5{}^{n+1}-u_5^n)$, que se conoce como 'relajación'. Después calcula $u'_6{}^{n+1}$ con Gauss-Seidel usando u_5^{n+1}, u_2, u_7^n, u_{10}^n, etc. w está entre 0 y 2 (si w=1 tenemos Gauss-Seidel),

existe un valor óptimo de w para cada ecuación. Para la ecuación de Laplace, $w=2/(1+sin(\Pi/N))$, donde N es el número de intervalos en la dirección Y.

- *SORCHECKERS*: Variación del método anterior, en el que se recorren los elementos de la matriz como si fuera un tablero de ajedrez: en una iteración las casillas blancas y en la otra las negras [Lapi82].

- *SSOR*: *SOR* simétrico, se recorre la matriz en una iteración de izquierda a derecha y de arriba abajo y en la siguiente al contrario [Lapi82].

Estas opciones se pueden indicar mediante la opción `ITERATIVE(<method>)`, donde `<method>` es uno de los cuatro métodos anteriores.

V.4.4 El método de las diferencias finitas explícito: esquema de DuFort-Frankel

La opción *DUFORTE* resuelve la *PDE* mediante el esquema de DuFort-Frankel, que es una modificación del esquema "*leapfrog*". El esquema es explícito e incondicionalmente estable, por ejemplo para la ecuación del calor no homogénea, si bien es un esquema no disipativo. Este esquema consiste en la modificación de la sustitución para la segunda derivada:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{U(r+1,s,t)-(U(r,s,t+1)+U(r,s,t-1))+U(r-1,s,t)}{h^2}$$

$$\frac{\partial^2 u}{\partial y^2} \approx \frac{U(r,s+1,t)-(U(r,s,t+1)+U(r,s,t-1))+U(r,s-1,t)}{k^2}$$

Ecuaciones V.28: Esquema DuFort - Frankel.

Con errores de truncamiento $O(h^2)$ y $O(k^2)$ como en los otros esquemas. Como toma información de dos niveles de tiempo (calcula *t+1* en función de *t* y *t-1*), ha de ser inicializado con otro esquema. Por ejemplo, para la ecuación $u_t+u_{xx}=0$, quedaría la siguiente molécula computacional (con un esquema centrado en el tiempo):

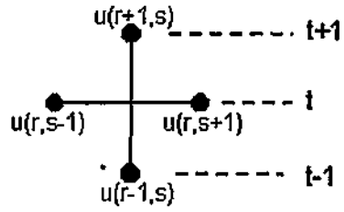


Figura V.24: Molécula computacional del esquema DuFort-Frankel para $u_t+u_{xx}=0$.

Si bien el esquema es explícito e incondicionalmente estable, sólo es consistente si *k/h* tiende a cero con *h* y *k* [Strik89].

La sintaxis es la siguiente:

```
PDE <id> (   <Ctt>, <ctt>, <Ct>,
             <Ax>, <ax>, <Ay>, <ay>, <Axy>, <axy>, <Ayx>, <ayx>,
             <Bx>, <By>, <a0>, <f>,
             DUFORTE )
```

Sintaxis V.17: Declaración de una *PDE*, resolución mediante el esquema de DuFort-Frankel.

■ V.4.5 El método de las diferencias finitas implícito: *IMPLICIT*

Este método resuelve la(s) ecuacione(s) mediante el método de Crank-Nicolson , para el caso de una dimensión, o mediante *ADI* (esquema Peaceman-Rachford [Peac55]), para el caso de dos dimensiones. La sintaxis es la siguiente:

```
PDE <id>  (<Ctt>, <ctt>, <Ct>,
           <Ax>, <ax>, <Ay>, <ay>, <Axy>, <axy>, <Ayx>, <ayx>,
           <Bx>, <By>, <a0>, <f>,
           IMPLICIT )
```

Sintaxis V.18: Declaración de una *PDE*, resolución mediante un método implícito.

El método de Crank-Nicolson consiste en centrar el esquema en diferencias alrededor de $t=(n+1/2)k$. De esta forma, se aplicarían las siguientes sustituciones [Stri89]:

$$\frac{\partial u}{\partial t} \approx \frac{u(t+1,r)-u(t,r)}{\Delta t}, O(\Delta t^2)$$

$$\frac{\partial^2 u}{\partial x^2} \approx 0.5\frac{u(t+1,r+1)-2u(t+1,r)+u(t+1,r-1)}{h^2}+0.5\frac{u(t,r+1)-2u(t,r)+u(t,r-1)}{h^2}, O(h^2)$$

$$f(t,r) \approx 0.5(f(t+1,r)+f(t,r))$$

Ecuaciones V.29: Discretizaciones para el esquema de Crank-Nicolson.

Por ejemplo, para la ecuación $u_t+u_{xx}=0$, dejaría la siguiente molécula computacional:

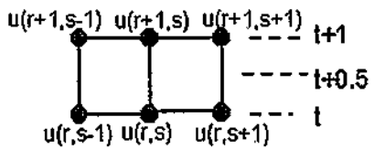


Figura V.25: Molécula computacional del esquema Crank-Nicolson para $u_t+u_{xx}=0$.

El método de Crank-Nicolson es implícito, ya que, como se ve en la figura anterior, no es posible despejar un nodo en la fila *t+1* en función sólo de nodos ya calculados, de líneas inferiores. También es incondicionalmente estable, y de precisión de orden (2,2).

El método *ADI* (Alternating Direction Implicit) fue introducido por Gourlay [Gour70]. Este método introdujo una nueva forma de descomponer el problema en dos partes más simples. Para el caso de dos dimensiones, el problema se reduce a varios problemas en una dimensión. Por ejemplo, sea la ecuación parabólica en dos dimensiones:

$$u_t = b_1 u_{xx} + b_2 u_{yy}$$

Ecuación V.30: Ecuación parabólica ejemplo.

Podemos definir operadores lineales A_1 y A_2 para la ecuación anterior:

$$A_1 u = b_1 u_{xx}$$
$$A_2 u = b_2 u_{yy}$$

Ecuaciones V.31: Operadores lineales para el ejemplo.

Y suponemos que tenemos métodos (tales como el de Crank-Nicolson) para resolver las ecuaciones en una dimensión:

$$w_t = A_1 w$$
$$w_t = A_2 w$$

Ecuaciones V.32: Ecuaciones en una dimensión asociadas con la ecuación V.30.

El método consiste en la misma idea que el de Crank-Nicolson, centrar el esquema en *t=(n+1/2)k*, con lo que se obtiene, para la ecuación V.30:

$$\frac{u(t+1,r,s)-u(t,r,s)}{\Delta t} = 0.5\cdot(A_1 u(t+1,r,s)+A_1 u(t,r,s))+0.5\cdot(A_2 u(t+1,r,s)+A_2 u(t,r,s))+O(\Delta t^2)$$

Ecuación V.33: Discretización de la ecuación V.30.

Pasando los términos en *t+1* a la izquierda, y multiplicando por *Δt*:

$$(I-\frac{\Delta t}{2}A_1-\frac{\Delta t}{2}A_2)u(t+1,r,s) = (I+\frac{\Delta t}{2}A_1+\frac{\Delta t}{2}A_2)u(t,r,s)+O(\Delta t^3)$$

Ecuación V.34

De acuerdo con la fórmula:

$$(1\pm a_1)(1\pm a_2) = 1\pm a_1 \pm a_2 + a_1 a_2$$

Ecuación V.35

y sumando $\Delta t^2 A_1 A_2 u(t+1,r,s)/4$ a ambos lados de V.33, tenemos:

$$(I-\frac{\Delta t}{2}A_1-\frac{\Delta t}{2}A_2+\frac{\Delta t^2}{4}A_1A_2)u(t+1,r,s) = (I+\frac{\Delta t}{2}A_1+\frac{\Delta t}{2}A_2+\frac{\Delta t^2}{4}A_1A_2)u(t,r,s)+$$
$$\frac{\Delta t^2}{4}A_1A_2(u(t+1,r,s)-u(t,r,s))+O(\Delta t^3)$$

Ecuación V.36

Que podemos factorizar de acuerdo con la ecuación V.35:

$$(I-\frac{\Delta t}{2}A_1)(I-\frac{\Delta t}{2}A_2)u(t+1,r,s) = (I+\frac{\Delta t}{2}A_1)(I+\frac{\Delta t}{2}A_2)u(t,r,s)+\frac{\Delta t^2}{4}A_1A_2(u(t+1,r,s)-u(t,r,s))+O(\Delta t^3)$$

Ecuación V.37

Se puede eliminar el segundo término de la parte derecha, porque es del orden de $O(\Delta t^3)$, que es el orden de los errores que ya se han introducido, quedando:

$$(I-\frac{\Delta t}{2}A_1)(I-\frac{\Delta t}{2}A_2)u(t+1,r,s)=(I+\frac{\Delta t}{2}A_1)(I+\frac{\Delta t}{2}A_2)u(t,r,s)$$

Ecuación V.38

En *OOCSMP*, para resolver la ecuación anterior, se usa el esquema de Peacemann-Rachford [Stri89], que es :

$$(I-\frac{\Delta t}{2}A_i)\tilde{u}(t+1/2,r,s)=(I+\frac{\Delta t}{2}A_2)u(t,r,s)$$
$$(I-\frac{\Delta t}{2}A_2)u(t+1,r,s)=(I+\frac{\Delta t}{2}A_1)\tilde{u}(t+1/2,r,s)$$

Ecuación V.39: Esquema de Peacemann-Rachford

Los dos pasos del esquema alternan qué dirección es implícita y cual es explícita. Este esquema es incondicionalmente estable. Otros posibles esquemas para la solución de la ecuación V.38 son el de D'Yakonov, o el de Douglas-Rachford [Stri89].

■ V.4.6 El método de los elementos finitos: *FEM*

La opción *FEM* permite solucionar la(s) ecuacion(es) mediante el método de los elementos finitos. La sintaxis es la siguiente:

```
PDE <id> (   <Ctt>,<ctt>,<Ct>,
             <Ax>,<ax>,<Ay>,<ay>,<Axy>,<axy>,<Ayx>,<ayx>,
             <Bx>,<By>,<a0>,<f>,
             FEM
             [, LUMPING(<lump-mode>)]
             [, ALPHA (<alpha-value>) ]
             [, GAMMA (<gamma-value>) ] )
```

Sintaxis V.19: Declaración de una *PDE*, resolución mediante el método de los elementos finitos.

LUMPING indica el tipo de 'lumping' o 'diagonalización' [Hugh87] que se ha de realizar sobre la matriz de masa o de la de amortiguamiento. *<lump-mode>* puede tomar los valores:

- *NO_LUMPING* : No se realiza ningún tipo de 'lumping'.
- *ROW_SUM_LUMP* : Se realiza una suma de los elementos de cada fila, que va a parar a la diagonal.
- *PROP_LUMP* : Como el anterior, pero proporcional al elemento de la diagonal.

ALPHA y *GAMMA* son los parámetros *ALPHA* y *GAMMA* para la solución del sistema mediante el método de Newmark [Hugh87]. Los valores por defecto son: 0.5 y 0 respectivamente.

V.4.7 Mezcla de métodos de resolución

En *OOCSMP* es posible mezclar métodos de resolución al solucionar una ecuación. Para ello, hay que dividir el dominio y discretizar cada parte, asignándose una ecuación a cada malla. La ecuación puede ser la misma, pero cambiando el método de resolución. Si dos mallas contiguas usan el mismo método de resolución, se resuelve como un todo, ampliándose el sistema de ecuaciones resultante, o bien, para el caso de diferencias finitas explícitas, recorriéndose ambas mallas como si fuesen una sola, pero mayor.

En el caso de que el método de resolución sea distinto, se ha de tener cuidado de que en los bordes de las mallas no se produzcan discontinuidades debido, por ejemplo a la imposibilidad de calcular la última fila o columna de la malla, y tener que aplicar una condición de contorno numérica en dicha fila o columna. Esto se ve fácilmente con el siguiente ejemplo sencillo:

Supóngase que se quiere resolver la ecuación del transporte en 2-D en un rectángulo. La ecuación que gobierna este fenómeno es: $u_t+au_x+bu_y=0$. El transporte se realiza de izquierda a derecha y de abajo a arriba, suponiendo que $a>0$ y $b>0$. Por tanto las condiciones de contorno se han de especificar a la izquierda y abajo. El rectángulo se divide en dos zonas, la primera zona a la izquierda, se quiere resolver mediante un esquema explícito, la segunda zona mediante el método *ADI* (ver figura V.26). En el esquema explícito se elegiría un esquema backwards tanto en x como en y (ver figura V.27).

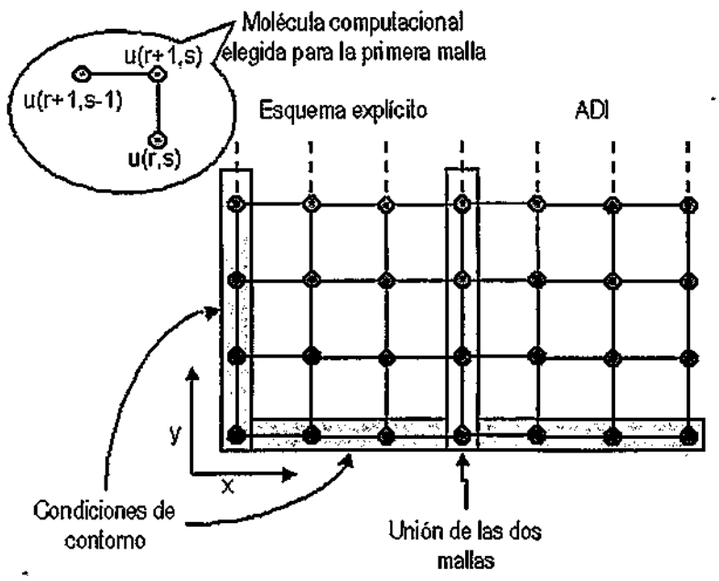


Figura V.26: Esquema del problema para solucionar $u_t+au_x+au_y=0$.

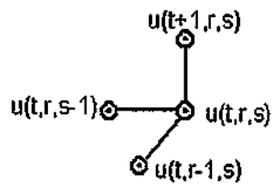


Figura V.27: Molécula computacional elegida para solucionar $u_t+au_x+au_y=0$.

El problema se empieza a solucionar por la parte izquierda (ya que es allí donde están las condiciones, y la ecuación transporta de izquierda a derecha), y el borde derecho de la primera zona se ha podido calcular. Cuando el método *ADI* empieza a solucionar, necesita el valor de la solución en el borde izquierdo de la segunda malla, que lo ha calculado el método anterior. En este caso, no hacen falta

condiciones de contorno numéricas en el borde de unión. Si por ejemplo, se eligiese para la primer zona un método que no pudiera calcular el último nodo de cada fila, se tendrían que imponer condiciones de contorno numéricas en el borde de unión, produciéndose un error en dicho borde. En la sección V.6.3 se presenta un ejemplo en el que ocurre este fenómeno.

Obviamente, para el ejemplo anterior no es una ventaja la mezcla de formas de resolución. Sí presentará ventajas este enfoque en problemas donde hubiera zonas en las que la ecuación variase poco, y se pudiera calcular mediante un esquema explícito, y zonas donde la solución tuviera más variación y requiriera un método más preciso, y más costoso en tiempo de cómputo, como es el de los elementos finitos.

V.5 Solución de sistemas y ecuaciones cuasilineales

Si se quiere resolver un sistema de ecuaciones, se debe declarar cada ecuación del sistema, y el primer parámetro del bloque *PDE*, ha de ser:

```
SYSTEM (<pde-1>, <pde-2> (,<pde-n>)* )
```

Sintaxis V.20: Primer parámetro del bloque *PDE* cuando declaramos un sistema de ecuaciones.

Es decir, se ha de indicar cada una de las ecuaciones que componen el sistema. Un ejemplo de declaración de un sistema de ecuaciones se presentó en la sección V.3.3, al declarar mallas elípticas.

Por ejemplo, supongamos que queremos resolver un problema de elasticidad plana, cuyas ecuaciones son:

$$h\rho\frac{\partial^2 u}{\partial t^2}-\frac{\partial}{\partial x}\left[h\left(c_{11}\frac{\partial u}{\partial x}+c_{12}\frac{\partial v}{\partial y}\right)\right]-\frac{\partial}{\partial y}\left[hc_{66}\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right]-f_x=0$$

$$h\rho\frac{\partial^2 v}{\partial t^2}-\frac{\partial}{\partial x}\left[hc_{66}\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right]-\frac{\partial}{\partial y}\left[h\left(c_{12}\frac{\partial u}{\partial x}+c_{22}\frac{\partial v}{\partial y}\right)\right]-f_y=0$$

Ecuaciones V.40: Ecuaciones de la elasticidad plana.

donde h es el grosor, ρ la densidad del material, u el desplazamiento horizontal y v el desplazamiento vertical. El sistema de ecuaciones se declararía de la siguiente forma en *OOCSMP*:

```
PDE displ_x( SYSTEM(displ_x, displ_y),
             h*p,1,0,0,                 *** Ctt, ctt,Ctt_2,ctt_2
             0,0,                       *** Ct, Ct_2
             -1,h*c11,0,0,              *** Ax, ax, Ax_2, ax_2
             -1,h*c66,0,0,              *** Ay, ay, Ay_2, ay_2
             0,0,-1,h*c12,              *** Axy, axy, Axy_2, axy_2
             0,0,-1,h*c66,              *** Ayx, ayx, Ayx_2, ayx_2
             0,0,0,0,0,0,-fx,FEM)       *** Bx_1,Bx_2,By_1,By_2,a0_1,a0_2,f

PDE displ_y( SYSTEM(displ_x, displ_y),
             0,0,h*p,1,                 *** Ctt, ctt,Ctt_2,ctt_2
             0,0,                       *** Ct, Ct_2
             0,0,-1,h*c66,              *** Ax, ax, Ax_2, ax_2
             0,0,-1,h*c22,              *** Ay, ay, Ay_2, ay_2
             -1,h*c66,0,0,              *** Axy, axy_1, Axy_2, axy_2
             -1,h*c12,0,0,              *** Ayx, ayx_1, Ayx_2,ayx_2
             0,0,0,0,0,0,-fy,FEM)       *** Bx_1,Bx_2,By_1,By_2,a0_1,a0_2,f
```

Ejemplo V.5 : Ecuaciones de la elasticidad plana expresadas en sintaxis *OOCSMP*.

Con *OOCSMP* también podemos resolver ecuaciones cuasilineales, en las que la función que estamos buscando aparezca en algún coeficiente que multiplique a los términos en derivadas. Por ejemplo, la ecuación del seno de Gordon, que es :

$$\Phi_{xx}-\Phi_{tt}=\text{sen}(\Phi)$$

Ecuación V.41 : Ecuación del seno de Gordon.

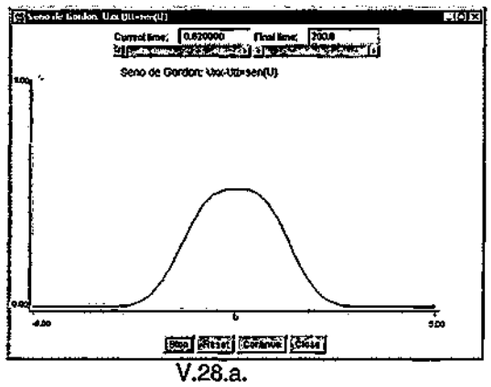
surge en varios fenómenos de materia condensada, y es una ecuación cuasilineal, ya que la función Φ aparece dentro se la función seno. Algunas de las soluciones de esta ecuación constituyen *solitones*, que son ondas que se propagan sin disiparse y tras chocar con otras semejantes no se destruyen ni se deforman. El modelo *OOCSMP* que resuelve la ecuación anterior se muestra en el listado V.1. Se han

incluido comentarios que indican los pasos del procedimiento de la figura V.10, el paso 4 (concatenar mallas), se ha omitido, ya que el dominio es simplemente una línea.

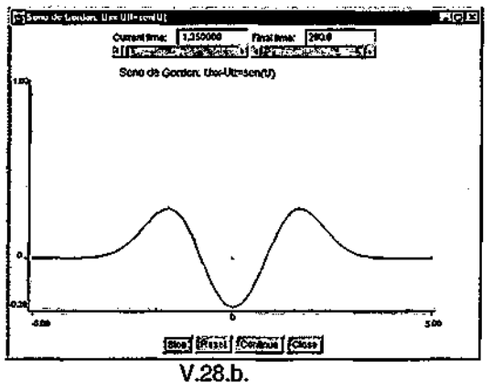
```
TITLE Seno de Gordon: Uxx-Utt=sen(U)
DOMAIN barId:=BAR(-5.0,5.0,                                          * Paso1
        INITIAL(EXP(-X*X)),INITIALDT(0.0), PERIODIC(EDGE(1,2)))      * Paso2
MESH Res := ISOPARAMETRIC(barId, LINE2, ELEMENTS(100) )              * Paso3
PDE SG(-1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -SIN(SG), DUFORT) * Paso5
Res.setPDE(SG)                                                       * Paso6
DYNAMIC
  Res.STEP()                                                         * Paso7
TIMER FINTIM := 20.0, delta := 0.01, PLdelta:= 0.01
PLOT2D [C], Res
```

Listado V.1 : Modelo de la ecuación del seno de Gordon en *OOCSMP*.

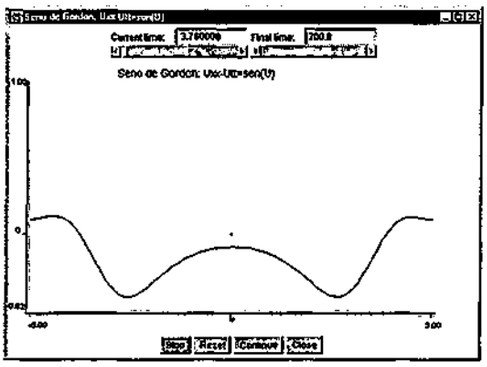
Como se puede observar, en el término que corresponde a la función independiente (parámetro número 15 del bloque *PDE*), hemos incluido una referencia a la propia *PDE* que estamos declarando (*SG*). En el caso de que estuvieramos resolviendo un sistema de ecuaciones, se podría referenciar a cualquiera de las ecuaciones que formaran parte del sistema. Hemos impuesto condiciones de contorno periódicas, y se ha perturbado inicialmente el dominio en e^{-x*x}. La solución del modelo anterior, en varios momentos, se puede observar en la secuencia de figuras V.28.



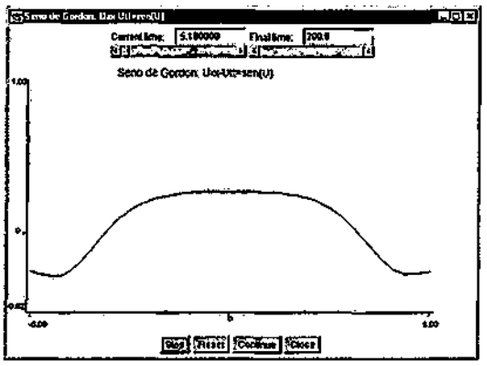
V.28.a.



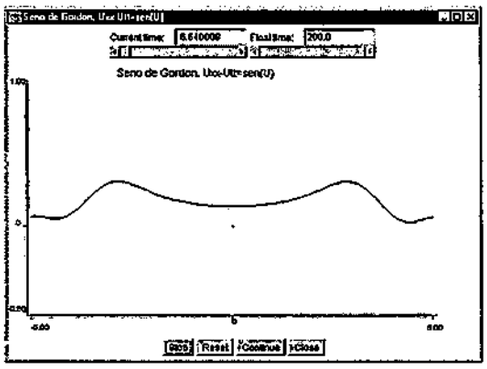
V.28.b.



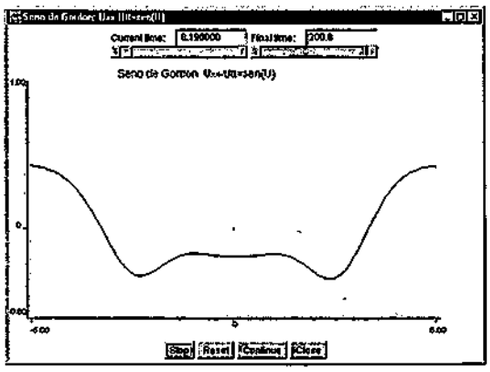
V.28.c.



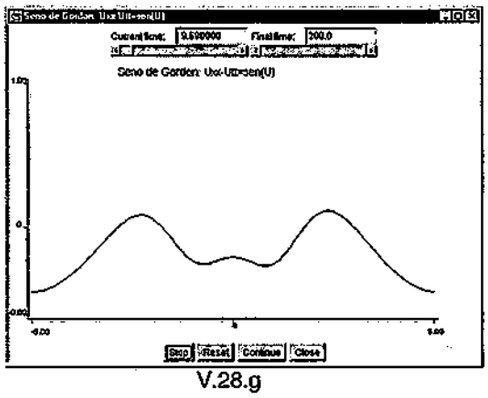
V.28.d.



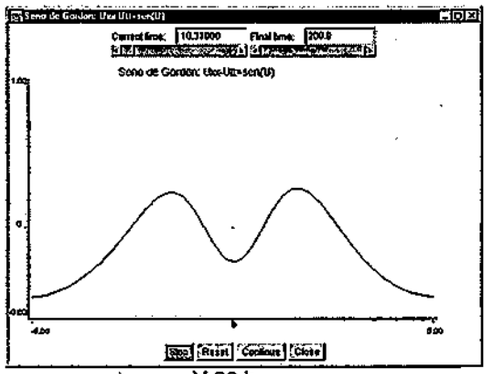
V.28.e.



V.28.f



V.28.g



V.28.h

Figuras V.28a-h: Distintos momentos en la resolución de la ecuación del seno de Gordon.

Otras ecuaciones cuasilineales que también producen soluciones solitónicas, como la ecuación $\Phi_{xx}-\Phi_{tt}=\lambda\Phi^3-m^2\Phi$, también son fácilmente representables mediante *OOCSMP*.

V.6 Ejemplos

En esta sección se presentan algunos ejemplos de resolución de *PDEs*. En algunos se muestran las ventajas del uso de *PDEs* junto con el paradigma de orientación a objetos, debido al ahorro de código, aumento de la claridad, y como se verá en el apartado VII, también existe la posibilidad de distribuir los objetos entre distintas máquinas. Otros ejemplos muestran la posibilidad de mezcla de distintos métodos de resolución.

V.6.1 Cálculo del calor en 4 vigas

Este ejemplo nos muestra el uso de :

- Análisis transitorio de *PDEs* 2-D.
- *PDEs* dentro de objetos.
- Concatenación de mallas dentro de objetos.
- Parametrización de mallas (translación) dentro de objetos.

Supongamos que se quiere simular el calentamiento de cuatro vigas en forma de L de gran longitud. Este problema se puede reducir a dos dimensiones, haciendo un corte transversal de las vigas. La figura V.29 muestra un esquema del problema. El calor que se aplica al entorno de las vigas sigue la ecuación $e^{2t}\sin(x+y)\cosh(x+y)$, y se muestra como un mapa de isosuperficies en la figura V.29, de forma que el color azul (esquina inferior izquierda), son las temperaturas más frías, y el color rojo (esquina superior derecha) son las temperaturas más altas.

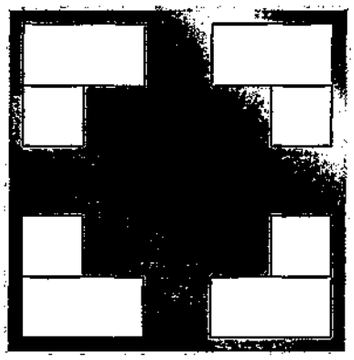


Figura V.29 : Esquema del problema a resolver

Al no ser simétrico el calor aplicado a cada una de las vigas, no podemos reducir el problema al cálculo de una sola viga. Pero es posible crear una clase (*LBeam*) que contenga la geometría de una viga en L genérica. Esta viga en L está formado por dos dominios primitivos, que son dos cuadriláteros, de longitud 2x1 y 1x1. A cada objeto de la clase *LBeam*, se le pasarán como parámetros la translación a aplicar a cada uno de los rectángulos para obtener la geometría del problema. En la clase *LBeam* también se discretiza la viga, y se le asigna la ecuación que queremos resolver, que es la ecuación del calor :

$$\frac{\partial F}{\partial t} + \frac{\partial F}{\partial x}\left(- Kx \frac{\partial F}{\partial x}\right) + \frac{\partial F}{\partial y}\left(- Ky \frac{\partial F}{\partial y}\right) = 0$$

Ecuación V.42 : Ecuación del calor en dos dimensiones.

El listado V.2 muestra el modelo *OOCSMP* para el problema, y la figura V.29, la solución. Existe una versión simplificada (las dos vigas de la parte superior, aunque también aparece un mapa de isosuperficies del calor aplicado) de este problema como parte de uno de los cursos para Internet que se presentan en el capítulo X.

Como puede observarse, la clase *LBeam* también recibe como parámetros la conductividad de la viga *Kx* y *Ky*. Como forma de visualización de la solución se ha elegido un mapa de isosuperficies para cada viga. Las salidas gráficas se detallan en el apartado IX.

```
TITLE Simulation of the heating of 4 pieces

APPHEAT X, Y
  APPHEAT:=EXP(2.0*TIME)*SIN(X+Y)*CH(X+Y)

CLASS Piece
{
    * Define class parameters, translation of the piece components
    DATA trx1, try1, trx2, try2
    * Conductivity of the material
    DATA Kx, Ky

    DOMAIN qd1 := QUADRILATERAL(0, 0, 2, 0, 2, 1, 0, 1
                  , INITIAL(0)
                  ,ESSENTIAL(EDGE(1:4),APPHEAT(X,Y))
                 )
    DOMAIN qd2 := QUADRILATERAL(0,-1, 1,-1, 1, 0, 0, 0
                  , INITIAL(0)
                  , ESSENTIAL(EDGE(1:4), APPHEAT(X,Y))
                 )

    qd1.MOVE (trx1, try1)
    qd2.MOVE (trx2, try2)

    * Mesh the domains
    MESH m1 := ISOPARAMETRIC (qd1, QUADRILAT4, ELEMENTS(40, 20) )
    MESH m2 := ISOPARAMETRIC (qd2, QUADRILAT4, ELEMENTS(20,20) )

    * Define the Heat equation in 2d
    PDE H2da(0, 0, 1, 1,-Kx, 1, -Ky, 0, 0, 0, 0, 0, 0, 0, 0, FEM )
    PDE H2db(0, 0, 1, 1,-Kx, 1, -Ky, 0, 0, 0, 0, 0, 0, 0, 0, FEM )
    m1.CONCAT(m2)
    m1.setPDE(H2da)
    m2.setPDE(H2db)

    DYNAMIC
    m1.STEP()
}

Piece p1 (0, 0, 0, 0, 3.0, 3.0 )
Piece p2 (3, 0, 4, 0, 3.5, 4.0 )
Piece p3 (0, -4, 0, -2, 3.0, 3.4 )
Piece p4 (3, -4, 4, -2, 3.1, 3.1 )


DYNAMIC
  Piece.STEP()

TIMER FINTIM:=1.0, delta:=0.05,PLdelta:=0.1
ISOPLOT   [C],  p1.m1
ISOPLOT   [E],  p2.m1
ISOPLOT   [S],  p3.m1
ISOPLOT   [SE], p4.m1
```

Listado V.2: Modelo *OOCSMP* del problema de las cuatro vigas

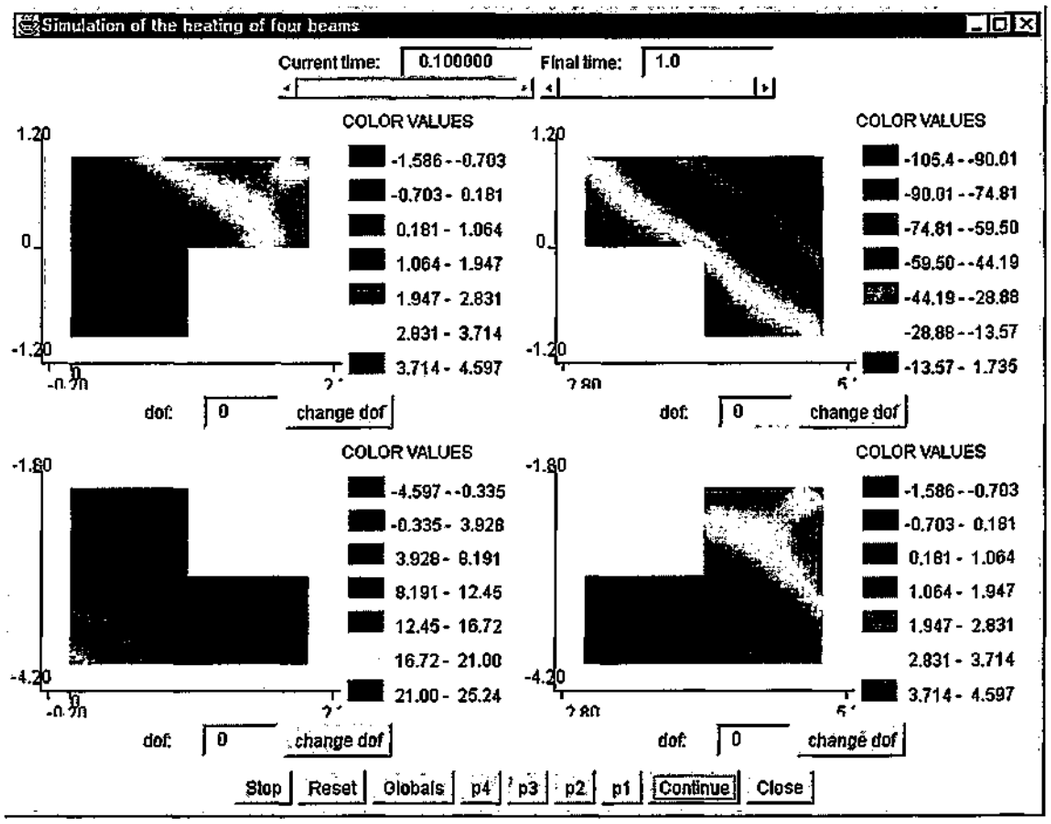


Figura V.30 : Un momento de la solución del problema V.6.1.

Con el uso de la orientación a objetos, se ha reducido notablemente la cantidad de código necesario para solucionar el problema, se ha mejorado la claridad, la extensibilidad y la posibilidad de reutilización. Más ventajas se obtienen con el uso de las extensiones para la distribución (ver capítulo VII). Este problema ha servido de base para el más complejo de las piezas móviles, que será presentado en el apartado sobre distribución del capítulo VII.

V.6.2 Cálculo del calor en una barra

Este ejemplo nos muestra el uso de :

- Análisis transitorio de *PDEs* 1-D.
- *PDEs* dentro de objetos.
- Concatenación de mallas de distintos objetos.
- Parametrización de mallas (número de elementos) dentro de objetos.

Supongamos que se quiere simular el calentamiento de una barra que tiene distintos coeficientes de conducción. La ecuación del calor en una dimensión viene dada por :

$$\frac{\partial u}{\partial t} - \frac{\partial}{\partial x}\left(K \frac{\partial u}{\partial x} \right) = 0$$

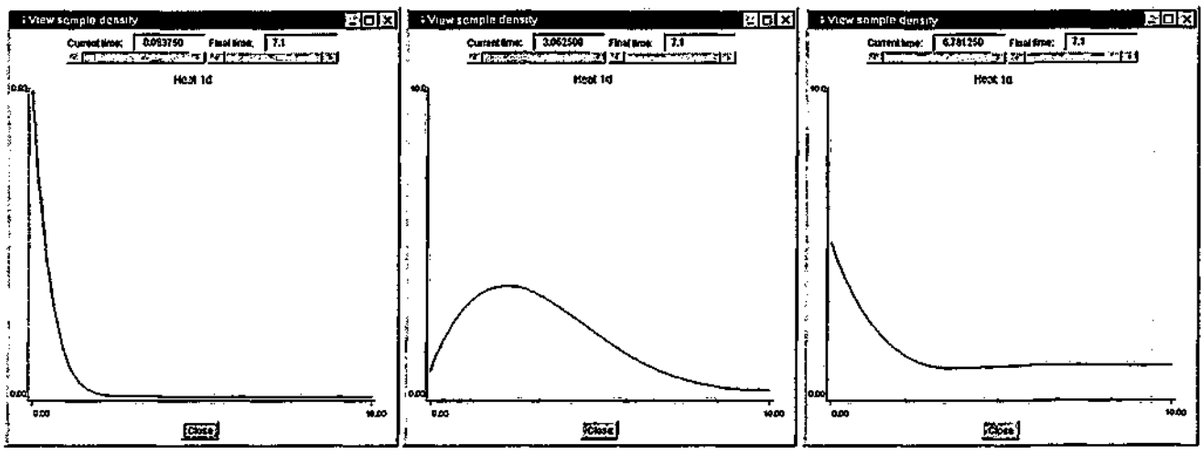
Ecuación V.43 : Ecuación del calor en una dimensión.

También queremos generar una malla no uniforme respecto a la distribución de nodos. Podemos modelar el problema encapsulando una porción de la barra en una clase. Cada objeto de la clase se parametrizará con sus dimensiones, los coeficientes de conducción, y el número de elementos de la malla. El listado V.3 muestra el modelo *OOCSMP*.

```
TITLE Heat equation in 1d (d/dt)u-K*(d2/dxx)u = 0
* Declare Conductivity coefficients
CLASS Bar
{
   NAME name
   * Dimensions of the bar
   DATA Xinit:=0.0 , Xend:=2.5
   * Conductivity coefficients
   DATA K := 4.8
   * number of elements
   DATA numElements := 50
   * Declare a piece of the bar, the default left BC is a senoid wave
   DOMAIN 1d:=BAR( Xinit,Xend,INITIAL(0.0),
                   ESSENTIAL(EDGE(1),SIN(TIME*100)))
   * Mesh the domain
   MESH   m :=ISOPARAMETRIC(1d,  LINE2, ELEMENTS(numElements))
   * declare the equation t be solved
   PDE tH (0,0,1,1,-K,0,0,0,0,0,0,0,0,0,IMPLICIT)
   * assign the equation to the mesh
   m.setPDE (tH)
}
* First bar takes all its parameters by default
Bar Bar1 ("Piece 1")
Bar Bar2 ("Piece 2",2.5, 5.0, 3.0)
Bar Bar3 ("Piece 3",5.0, 10.0, 2.5, 100)
* This overwrites the default left BC for Bars 2 and 3
Bar1.m.CONCAT (Bar2.m , Bar3.m)
DYNAMIC
Bar1.m.STEP()
PLOT Bar1.m
* Define the time increment, and the final time
TIMER  delta:=0.01, FINTIM:=5.0
```

Listado V.3: Simulación del calor en una barra.

En este ejemplo, el primer pedazo de la barra se divide en 50 elementos, el segundo también, pero el coeficiente de conducción cambia a 2.5, el tercer trozo se discretiza con 100 elementos. Las siguientes figuras muestran una evolución de la solución.



(a) (b) (c)

Figuras V.31a,b y c: Solución del problema V.6.2 en distintos momentos.

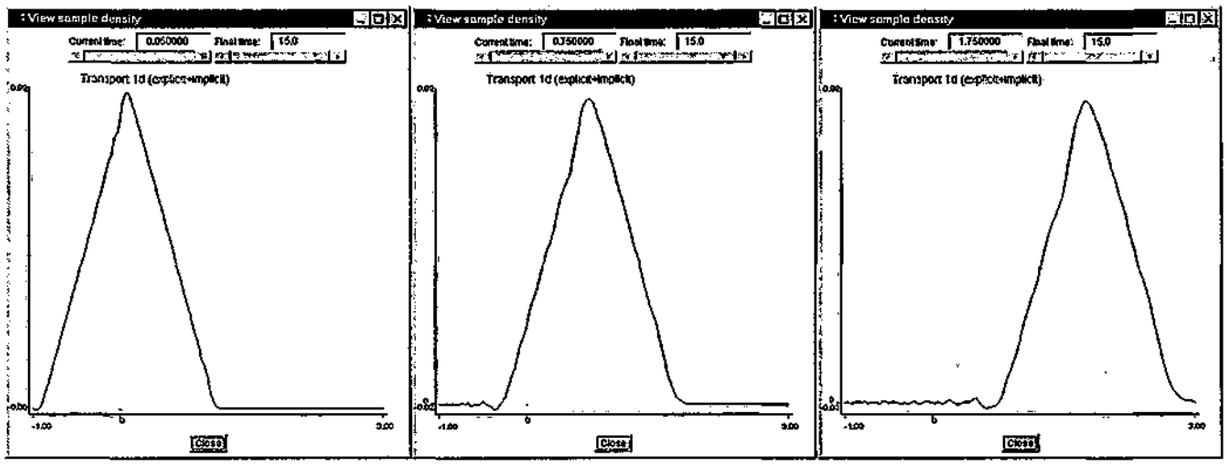
■ V.6.3 Ecuación del transporte en 1D

En el siguiente problema vamos a resolver la ecuación del transporte en una dimensión mezclando dos esquemas de diferencias finitas. La molécula computacional explícita más adecuada se discutió en la sección V.4.1. El dominio del problema lo dividiremos en dos, en la primera parte, solucionaremos mediante un esquema explícito (*backwards* en el espacio), el segundo tramo mediante Crank-Nicolson. Al ser el primer esquema backwards, no va a haber problemas en la unión de las mallas, no se va a necesitar una condición de contorno numérica. El listado es el siguiente :

```
TITLE TRANSPORT EQUATION IN 1D
DATA A:=1
DOMAIN 11 := BAR(-1.0, 1.0,
                 INITIAL(FCNSW(ABS(2-X)-1, 1-ABS(2-X), 1-ABS(2-X), 0)),
                 ESSENTIAL (EDGE(1),0.0) )
DOMAIN 12 := BAR(1.0, 3.0,
                 INITIAL(FCNSW(ABS(2-X)-1, 1-ABS(2-X), 1-ABS(2-X), 0)))

MESH m1 := ISOPARAMETRIC(11,LINE2,ELEMENTS(50))
MESH m2 := ISOPARAMETRIC(12,LINE2,ELEMENTS(50))

m1.CONCAT (m2)

PDE trp1 (0,0,1,0,0,0,0,0,0,0,0,A,0,0,0,EXPLICIT,XScheme(BACKWARDS))
PDE trp2 (0,0,1,0,0,0,0,0,0,0,0,A,0,0,0,IMPLICIT)

m1.setPDE(trp1)
m2.setPDE(trp2)
DYNAMIC
  m1.STEP()
PLOT m1
TIMER delta:= 0.0025, PRdelta:= 0.05, FINTIM:=15
```

Listado V.4: Problema del transporte en 1-D.

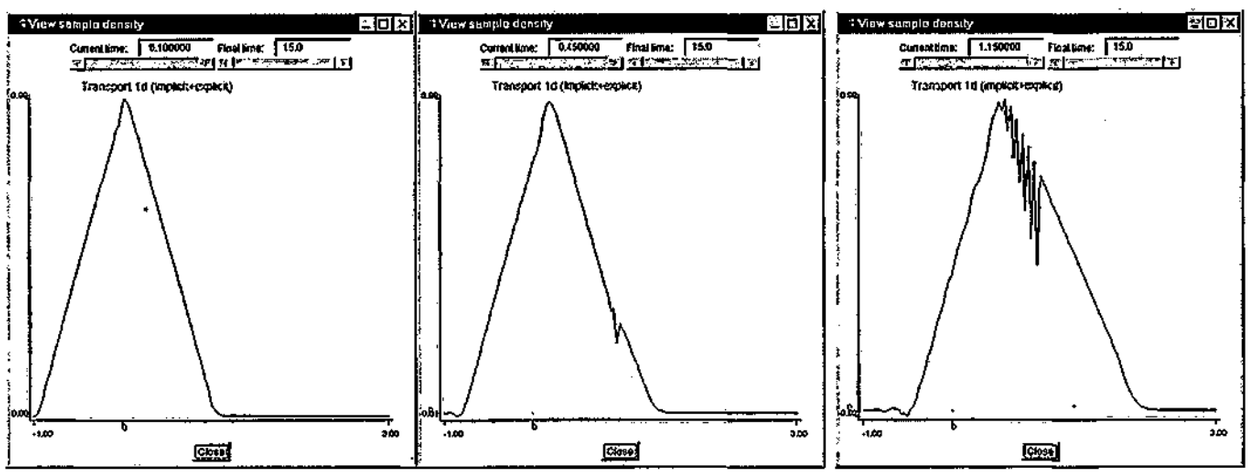
La solución en distintos momentos de tiempo se muestra en las siguientes figuras:



(a) (b) (c)

Figuras V.32a, b y c: Solución del problema V.6.3 en distintos momentos.

Supongamos ahora que se soluciona al revés: la primera parte del dominio con un método implícito, y la segunda mediante un método explícito. En este caso se va a tener que aplicar una condición de contorno numérica en la unión. En las siguientes figuras, se puede observar dicho error, que se propaga con el tiempo a la totalidad de la primera malla.



(a) (b) (c)

Figuras V.33a,b y c: Solución del problema V.6.3 en distintos momentos, mezcla errónea de métodos de solución.

V.7 *MGEN*

V.7.1 Introducción

En esta sección se va a describir nuestra herramienta gráfica de generación de mallas *MGEN* (Mesh GENerator). Esta herramienta se ha construido usando el lenguaje Java, lo que nos va a permitir usarla a través de Internet dentro de una simulación. Si bien, como ya se dijo en la sección V.1, también se puede usar como un programa separado, como generador de código *OOCSMP*.

Los elementos de la interfaz de *MGEN* (cuando se utiliza como generador de código) se muestran en la figura 1.

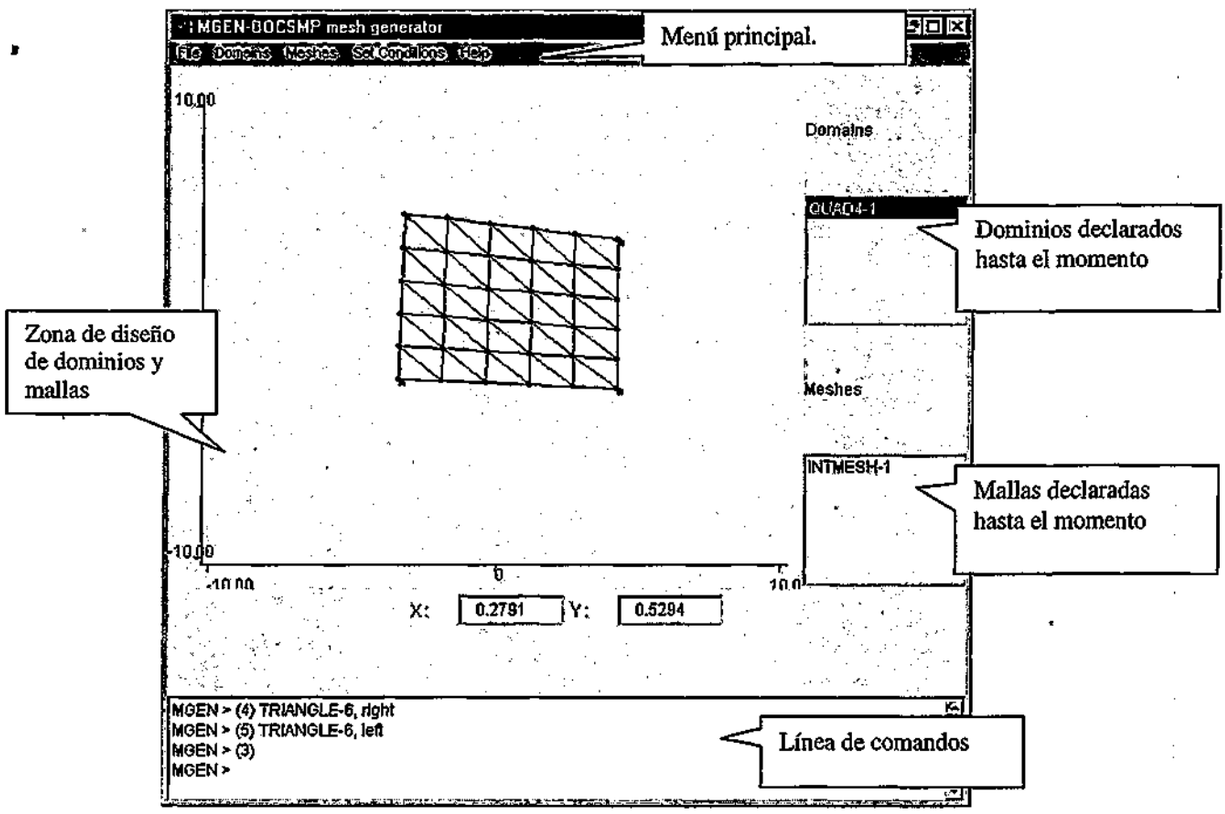


Figura V.34: Interfaz de *MGEN*

La herramienta puede controlarse mediante comandos que pueden introducirse en el campo de edición de la zona inferior, o mediante el menú desplegable y acciones del ratón.

V.7.2 *MGEN* como generador de código *OOCSMP*

MGEN se comporta como un generador de código *OOCSMP* cuando se ejecuta como un programa Java independiente. Esta forma de ejecutar *MGEN* no permite la inclusión de funciones *OOCSMP* dinámicas como condiciones iniciales o de contorno (ver sección V.2). Aunque sí permite grabar y cargar archivos con sintaxis *OOCSMP*.

Las acciones que se pueden realizar con *MGEN* son las siguientes (agrupadas por su lugar en el menú principal):

File	Domains	Meshes	Set Conditions	Help
Save	Circular Sector	Delaunay Mesh	Boundary	About
Load	Quadrilateral 4	Interpolation Mesh	Initial	Meshing with *MGEN*
Show *OOCSMP* code	Quadrilateral 8	Elliptic Mesh		
Zoom In	Triangle	Concatenate		
Zoom Out	Move	Smooth		
Redraw all	Rotate			
Exit	Scale			
	Clear			

Tabla V.6: Acciones posibles del menú de *MGEN*

A continuación se explica cada una de estas acciones con detalle.

V.7.2.1 Menú "File"

V.7.2.1.1 Acción File: Save

Esta acción es accesible desde el menú *File*, o también mediante la instrucción *SAVE* introducida en la línea de comandos y muestra un cuadro de diálogo en el que se ha de elegir el nombre da archivo a guardar. El fichero se guarda con la sintaxis *OOCSMP* de los dominios, mallas y condiciones iniciales y de contorno que se hayan diseñado dentro de *MGEN*.

V.7.2.1.2 Acción File: Load

Esta acción es accesible desde el menú *File*, o también mediante la instrucción *LOAD* introducida en la línea de comandos. Muestra un cuadro de diálogo en el que se ha de elegir el nombre del archivo a cargar. El fichero a cargar debe tener la sintaxis de *OOCSMP*.

V.7.2.1.3 Acción File: Show *OOCSMP* code

Esta acción es accesible desde el menú *File*, o también mediante la instrucción *SYNTAX* de la línea de comandos. Esta acción muestra en la zona de edición la sintaxis *OOCSMP* de todo lo que se ha diseñado hasta el momento.

V.7.2.1.4 Acción File: Zoom in

Esta acción es accesible desde el menú *File*, o también mediante el mandato *ZOOMIN* de la línea de comandos. Permite agrandar una zona del espacio de diseño. Para lo cual, se debe pinchar con el ratón en un lugar de la zona de diseño y arrastrar hasta conseguir un rectángulo que englobe la zona que se quiere agrandar.

V.7.2.1.5 Acción File: Zoom out

Esta acción es accesible desde el menú *File*, o también mediante el mandato *ZOOMOUT* de la línea de comandos. Permite aumentar (y también disminuir) la escala de la zona de diseño. Para ello hay que colocar los valores máximos y mínimos de los ejes, en una ventana emergente. El aspecto de dicha ventana se muestra en la Figura V.35.

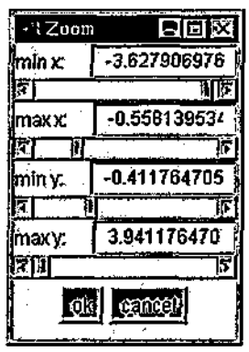


Figura V.35: Ventana de *Zoom Out*.

V.7.2.1.6 Acción File: Redraw All

Esta acción es accesible desde el menú *File*, o también mediante el mandato *REDRAW* de la línea de comandos y redibuja todo el contenido de la zona de diseño.

V.7.2.1.7 Acción File: Exit

Esta acción es accesible desde el menú *File*, o también mediante el mandato *EXIT* de la línea de comandos, y sirve para salir del programa.

V.7.2.2 Menú "Domains"

V.7.2.2.1 Acción Domains: Circular sector

Esta acción es accesible desde el menú *Domains*, o también mediante la instrucción *CIRCSECTOR* de la línea de comandos. Se corresponde con la primitiva de dominio *OOCSMP* permite definir un sector circular. Como respuesta a esta acción, en la zona de edición, se visualizará el siguiente mensaje:

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > (
```

Listado V.5: Primer paso en la definición de un sector circular.

Ante el cual, hay que introducir el centro del sector circular pinchando en la zona de diseño, o mediante el teclado. Si se introducen mediante el teclado, hay que introducir las dos coordenadas separadas por una coma, y al final un paréntesis cerrado. Como respuesta, se obtiene un mensaje similar al del siguiente listado. Por ejemplo, al introducir el centro en (0.0, 0.0) se obtiene:

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > ( 0.0, 0.0 )
MGEN > Input inner radius:
MGEN > (
```

Listado V.6: Segundo paso en la definición de un sector circular, centro.

Tras introducir el punto central, éste se queda señalado en la zona de diseño, y se nos pregunta por el radio interno. Se puede introducir también por medio del teclado (un número real seguido de un paréntesis cerrado), o por medio del ratón, en este último caso, se toma como el radio la distancia entre el centro y el punto indicado. Siguiendo con el ejemplo :

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > ( 0.0, 0.0 )
MGEN > Input inner radius:
MGEN > ( 2.0 )
MGEN > Inner radius= 2.0000
MGEN > Input outer radius:
MGEN > (
```

Listado V.7: Tercer paso en la definición de un sector circular, radio interno.

Lo siguiente a introducir es el radio exterior, que se introduce de la misma forma que el radio interior :

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > ( 0.0, 0.0 )
MGEN > Input inner radius:
MGEN > ( 2.0 )
MGEN > Inner radius= 2.0000
MGEN > Input outer radius:
MGEN > ( 5.0 )
MGEN > Outer radius= 5.0000
MGEN > Input initial angle:
MGEN > (
```

Listado V.8: Cuarto paso en la definición de un sector circular, radio externo.

Ahora se ha de introducir el ángulo inicial del sector circular, si se introduce por medio del teclado, se ha de introducir en radianes :

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > ( 0.0, 0.0 )
MGEN > Input inner radius:
MGEN > ( 2.0 )
MGEN > Inner radius= 2.0000
MGEN > Input outer radius;
MGEN > ( 5.0 )
MGEN > Outer radius= 5.0000
MGEN > Input initial angle:
MGEN > ( 0.0 )
MGEN > Init angle= 0.0000
MGEN > Input final angle:
MGEN > (
```

Listado V.9: Quinto paso en la definición de un sector circular, ángulo inicial.

El ángulo final se introduce de igual manera :

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > ( 0.0, 0.0 )
MGEN > Input inner radius:
MGEN > ( 2.0 )
MGEN > Inner radius= 2.0000
MGEN > Input outer radius:
MGEN > ( 5.0 )
MGEN > Outer radius= 5.0000
MGEN > Input initial angle:
MGEN > ( 0.0 )
MGEN > Init angle= 0.0000
MGEN > Input final angle:
MGEN > ( 1.7 )
MGEN > Final angle= 1.7000
```

Listado V.10: Listado completo para la definición de un sector circular, ángulo final.

Hecho lo cual, se muestra el dominio en la zona de diseño, y en la lista de dominios definidos. El resultado se muestra en la figura V.36.

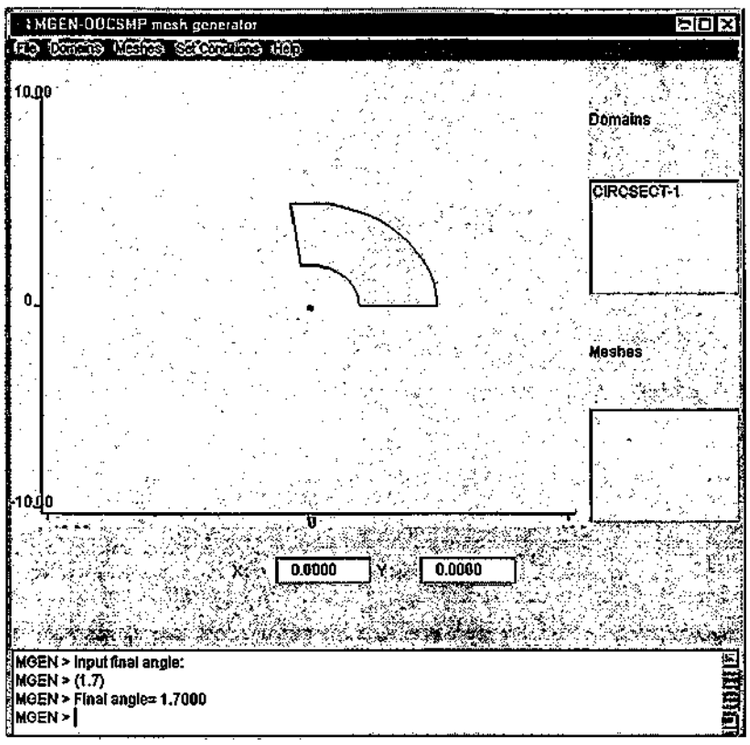


Figura V.36: Definición de un dominio sector circular.

V.7.2.2.2 Acción Domains: Quadrilateral 4

Esta acción es accesible desde el menú *Domains*, o también mediante la instrucción *QUADRILAT4* de la línea de comandos. Se corresponde con la primitiva de dominio *OOCSMP* que permite definir un cuadrilátero. Para definir un dominio de este tipo, se han de indicar los cuatro vértices empezando por el inferior izquierdo, y en sentido contrario a las agujas del reloj. El siguiente listado es la secuencia para la definición de un cuadrilátero :

```
MGEN > QUADRILAT4
MGEN > Define Quadrilat 4.
MGEN > Input lower left point :
MGEN > (-5,-5)
MGEN > first point at (-5.0000,-5.0000)
MGEN > Input lower right point :
MGEN > (5,-5)
MGEN > second point at ( 5.0000,-5.0000)
MGEN > Input upper right point :
MGEN > (7,5)
MGEN > third point at ( 7.0000, 5.0000)
MGEN > Input upper left point :
MGEN > (-3,5)
MGEN > fourth point at (-3.0000, 5.0000)
```

Listado V.11: Definición de un cuadrilátero de 4 nodos.

Y el resultado visual de esta operación se muestra en la siguiente figura:

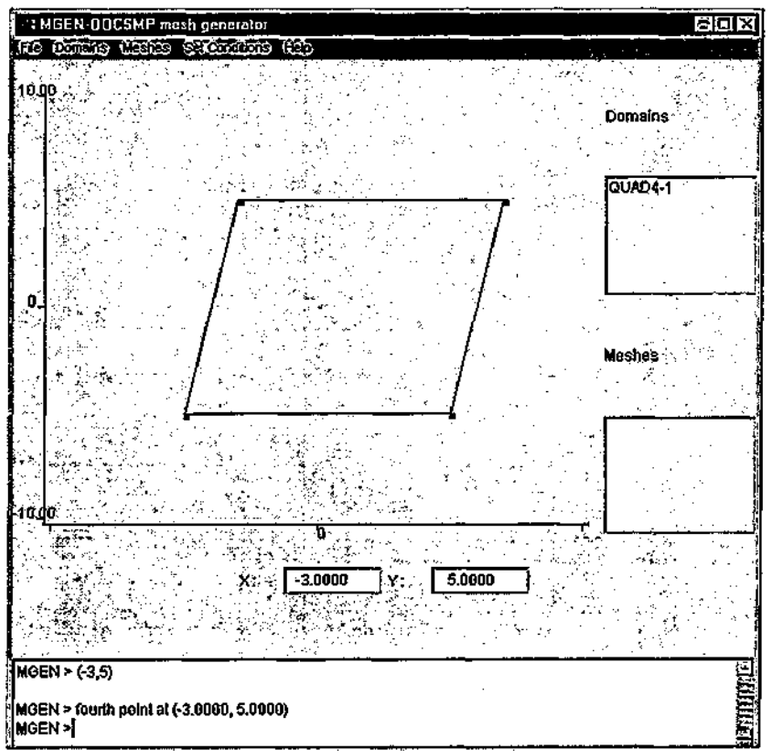


Figura V.37: Definición de un cuadrilátero.

V.7.2.2.3 Acción Domains: Quadrilateral 8

Esta opción permite definir un cuadrilátero de lados curvos (parábolas). Se puede acceder a esta opción mediante el menú *Domains*, o introduciendo la instrucción *QUADRILAT8* en la zona de edición. Para definir un dominio de este tipo, se ha de indicar la posición de los 8 puntos que lo componen. Cada lado del dominio, está formado por tres puntos, siendo por tanto, parábolas cada uno de ellos. El listado V.12 es un ejemplo de la definición de un cuadrilátero de este tipo. La figura V.38 muestra el resultado de esta secuencia.

```
MGEN >QUADRILAT8
MGEN > Define 8 point quadrilateral.
MGEN > Input first point :
MGEN > (-5,-5)
MGEN > first point at (-5.0000,-5.0000)
MGEN > Input second point :
MGEN > (0,-7)
MGEN > second point at ( 0.0000,-7.0000)
MGEN > Input third point :
MGEN > (5,-5)
MGEN > third point at ( 5.0000,-5.0000)
MGEN > Input fourth point :
MGEN > (3,0)
MGEN > fourth point at ( 3.0000, 0.0000)
MGEN > Input fifth point :
MGEN > (5,5)
MGEN > fifth point at ( 5.0000, 5.0000)
MGEN > Input sixth point :
MGEN > (0,7)
MGEN > sixth point at ( 0.0000, 7.0000)
MGEN > Input seventh point :
MGEN > (-5,5)
MGEN > seventh point at (-5.0000, 5.0000)
```

```
MGEN > Input eight point :
MGEN > (-3,0)
MGEN > eight point at (-3.0000, 0.0000)
```

Listado V.12: Definición de un cuadrilátero de ocho nodos

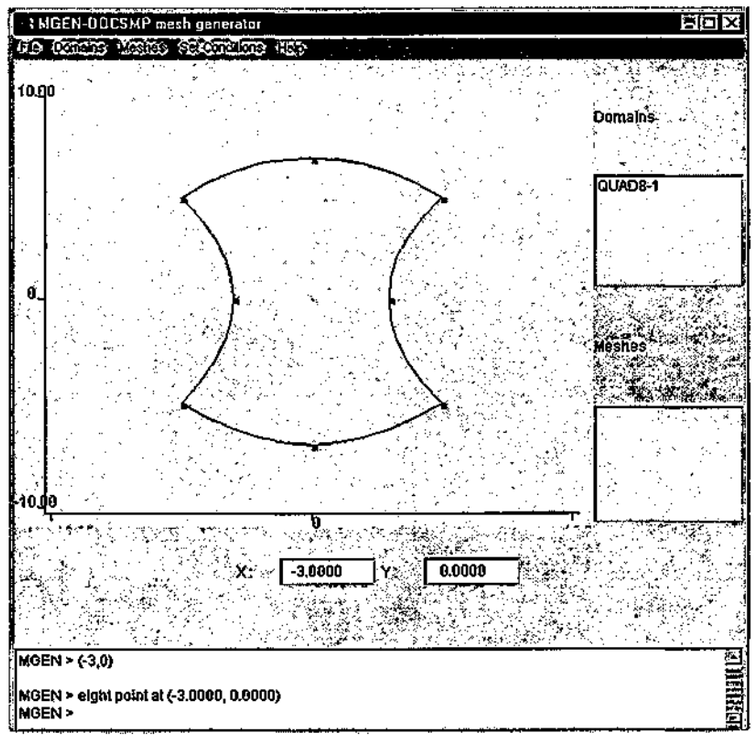


Figura V.38: Un cuadrilátero de lados curvos.

V.7.2.2.4 Acción Domains: Triangle

Esta opción permite definir un triángulo. Esto se puede efectuar mediante el menú *Domains*, o introduciendo la instrucción *TRIANGLE* en la zona de edición. Para definir un dominio de este tipo, se ha de indicar la posición de los tres puntos que lo componen en sentido antihorario, empezando por la esquina inferior izquierda. El listado V.13 es un ejemplo de la definición de un triángulo. La figura V.39 muestra el resultado de esta secuencia.

```
MGEN > TRIANGLE
MGEN > Define Triangle.
MGEN > Input lower left point :
MGEN > (-5,-5)
MGEN > first point at (-5.0000,-5.0000)
MGEN > Input second point :
MGEN > (5,-5)
MGEN > second point at ( 5.0000,-5.0000)
MGEN > Input third point :
MGEN > (0,7.071)
MGEN > third point at ( 0.0000, 7.0710)
```

Listado V.13: Definición de un triángulo

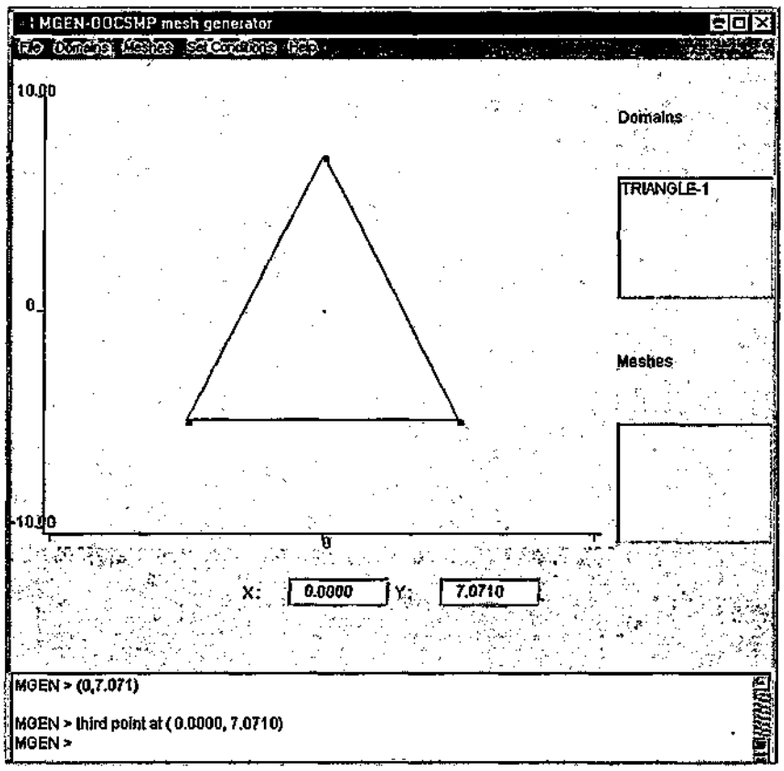


Figura V.39: Un triángulo.

V.7.2.2.5 Acción Domains: Move

Esta opción permite transladar un dominio (y su correspondiente malla, si la hubiera). Se accede mediante el menú *Domains*, o mediante el mandato *MOVE*.

Lo primero que hay que hacer en este mandato es seleccionar el dominio que se quiere mover. Esto se puede hacer seleccionando el dominio de la lista *Domains*, o bien pinchando con el ratón en el interior de uno de los dominios de la zona de diseño. En la zona de edición se podrá ver entonces:

```
MGEN > Domain <DOMAIN-NAME> selected
MGEN > Enter X displacement
MGEN > (
```

Listado V.14: Translación de un dominio.

Donde *<DOMAIN-NAME>* es el nombre del dominio seleccionado. Ahora se tiene que introducir el desplazamiento horizontal, y luego el vertical. Se pueden introducir mediante el teclado, o bien pinchando con el ratón en un punto dentro del dominio, y arrastrando luego hasta la posición final (de esta forma se introduce el desplazamiento horizontal y el vertical a la vez).

V.7.2.2.6 Acción Domains: Rotate

Esta acción permite girar un dominio (y su correspondiente malla, si la hubiera) por un eje que pasa por su baricentro. Se accede mediante el menú *Domains*, o mediante el mandato *ROTATE*.

Lo primero que hay que hacer es, como en el caso anterior, seleccionar el dominio que hay que rotar. Una vez hecho esto, se ha de indicar los grados, en radianes. Un radio positivo, implica un giro en sentido antihorario.

V.7.2.2.7 Acción Domains: Scale

Esta acción permite escalar (aumentar o disminuir de tamaño) un dominio y su correspondiente malla (si la hubiera). Se accede mediante el menú *Domains*, o mediante el mandato *SCALE*.

Lo primero que hay que hacer es, como en el caso anterior, seleccionar el dominio que hay que escalar. Una vez hecho esto, se ha de indicar la magnitud para la escala. Un número menor que uno y mayor que cero, implica una reducción de escala, un número mayor que uno, un aumento.

V.7.2.2.8 Acción Domains: Clear

Esta acción permite borrar un dominio, y su correspondiente malla, si la hubiera. Se accede mediante el menú *Domains*, o mediante el mandato *CLEAR*.

V.7.2.3 Menú "Meshes"

V.7.2.3.1 Acción Meshes : Delaunay Mesh

Esta opción permite discretizar un dominio mediante la triangulación de Delaunay. Está accesible desde el menú Meshes, o bien mediante el mandato *DELAUNAY*.

Una vez introducido el comando, lo primero que hay que hacer es seleccionar el dominio que se quiere discretizar. Luego hay que introducir el número de elementos en la dirección *X* del dominio, y en la dirección *Y*. Más tarde, se debe seleccionar el tipo de símplice que se usará para rellenar el dominio, y las restricciones. Se puede añadir como máximo, una restricción de cada tipo (mínimo ángulo, área máxima de cada triángulo, máximo tamaño de los lados de los triángulos).

El siguiente ejemplo muestra la triangularización de un cuadrado de lado 10, con 5 elementos horizontales y 5 verticales. Se ha puesto una restricción de área máxima de 2.0 y de máximo tamaño de lado de 2.2. La secuencia para definir la malla y el dominio ha sido la siguiente:

```
MGEN > QUADRILAT4
MGEN > Define Quadrilat 4.
MGEN > Input lower left point :
MGEN > (-5,-5)
MGEN > first point at (-5.0000,-5.0000)
MGEN > Input lower right point :
MGEN > (5,-5)
MGEN > second point at ( 5.0000,-5.0000)
MGEN > Input upper right point :
MGEN > (5,5)
MGEN > third point at ( 5.0000, 5.0000)
MGEN > Input upper left point :
MGEN > (-5,5)
MGEN > fourth point at (-5.0000, 5.0000)
MGEN >DELAUNAY
MGEN > Select domain
MGEN > Selected domain QUAD4-1
MGEN > Select number of X elements
MGEN > (5)
MGEN > 5 X elements.
MGEN > Select number of Y elements
MGEN > (5)
MGEN > 5 Y elements.
MGEN > Select simplex
MGEN > (0) QUADRILAT-4
MGEN > (1) QUADRILAT-8
MGEN > (2) TRIANGLE-3
MGEN > (3) TRIANGLE-6
MGEN > (2)
MGEN > Simplex TRIANGLE-3 selected.
```

```
MGEN > Select constraints :
MGEN > (0) Angle constraint.
MGEN > (1) Area constraint.
MGEN > (2) Edge constraint.
MGEN > (3) No more constraints.
MGEN > (1)
MGEN > Input maximum area
MGEN > (2.0)
MGEN > Select constraints :
MGEN > (0) Angle constraint.
MGEN > (1) Area constraint.
MGEN > (2) Edge constraint.
MGEN > (3) No more constraints.
MGEN > (2)
MGEN > Input maximum edge size
MGEN > (2.2)
MGEN > Select constraints :
MGEN > (0) Angle constraint.
MGEN > (1) Area constraint.
MGEN > (2) Edge constraint.
MGEN > (3) No more constraints.
MGEN > (3)
```

Listado V.15: Triangularización de un cuadrilátero.

El resultado de la secuencia anterior se muestra en la figura V.40.

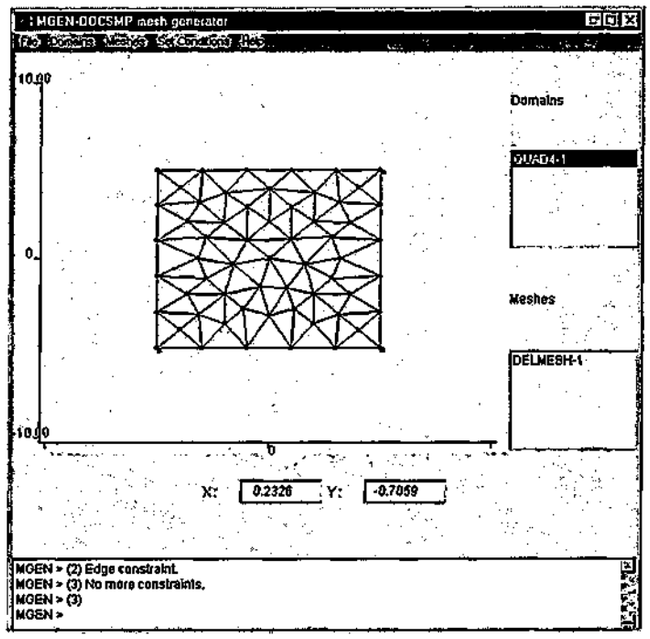


Figura V.40: Discretización de un cuadrilátero mediante Delaunay.

V.7.2.3.2 Acción Meshes : Interpolation Mesh

Esta acción permite discretizar un dominio mediante elementos isoparamétricos, excepto en el caso de un sector circular, que se hace mediante el paso de coordenadas cartesianas a polares.

Está accesible desde el menú Meshes, o bien mediante el mandato *ISOPARAMETRIC* en la línea de comandos.

La siguiente secuencia muestra la discretización de un sector circular. Se han usado 15 elementos (cuadrados de 4 nodos) en la dirección *X* y 5 en la dirección *Y*. En el caso del sector circular, la dirección *X* es la que rodea al sector interno y al externo, y la dirección *Y* es la radial.

```
MGEN > CIRCSECTOR
MGEN > Define Circular Sector.
MGEN > Input center point :
MGEN > (0.0,0.0)
MGEN > Center = ( 0.0000, 0.0000)
MGEN > Input inner radius:
MGEN > (2.0)
MGEN > Inner radius= 2.0000
MGEN > Input outer radius:
MGEN > (7.0)
MGEN > Outer radius= 7.0000
MGEN > Input initial angle:
MGEN > (0)
MGEN > Init angle= 0.0000
MGEN > Input final angle:
MGEN > (1.57)
MGEN > Final angle= 1.5700
MGEN > ISOPARAMETRIC
MGEN > Select domain
MGEN > Selected domain CIRCSECT-1
MGEN > Select number of X elements
MGEN > (15)
MGEN > 15 X elements.
MGEN > Select number of Y elements
MGEN > (5)
MGEN > 5 Y elements.
MGEN > Select simplex
MGEN > (0) QUADRILAT-4
MGEN > (1) QUADRILAT-8
MGEN > (2) TRIANGLE-3, right
MGEN > (3) TRIANGLE-3, left
MGEN > (4) TRIANGLE-6, right
MGEN > (5) TRIANGLE-6, left
MGEN > (0)
```

Listado V.16: Discretización de un Sector Circular.

El resultado de la secuencia anterior se muestra en la figura V.41.

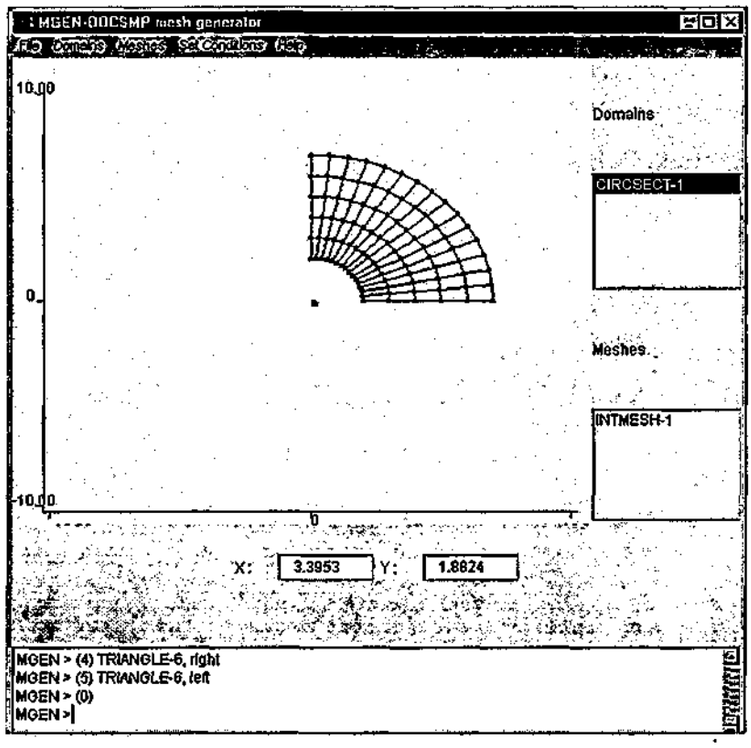


Figura V.41: Discretización de un Sector Circular mediante elementos isoparamétricos.

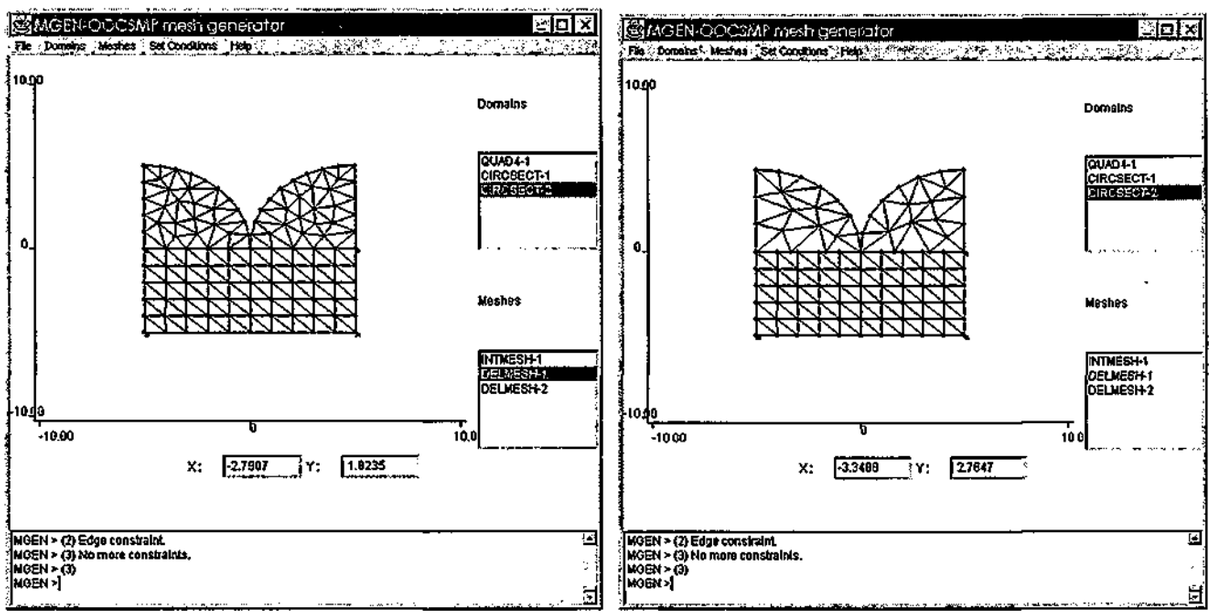
V.7.2.3.3 Acción Meshes: Elliptic Mesh

Esta permite discretizar un dominio mediante un generador elíptico. El generador usa la ecuación de *Laplace* para la discretización. Los pasos que hay que efectuar para una discretización de este tipo son los mismos que para la generación isoparamétrica, siendo el resultado, con las ecuaciones de Laplace, bastante similar.

V.7.2.3.4 Acción Meshes: Concatenate

Esta acción concatena dos mallas con al menos un punto en común. Está accesible desde el menú Meshes, o mediante el comando *CONCAT*. Si la operación tiene éxito, las dos mallas pasan a formar una sola malla compuesta.

En la mayoría de los casos, para que la operación tenga sentido, es necesario que los nodos generados en el borde común de ambas mallas coincidan. Las siguientes figuran muestran dos ejemplos de concatenación, en el primero, se ha hecho correctamente, ya que los nodos de los bordes coinciden. En el segundo caso, se ha hecho incorrectamente, ya que los nodos no coinciden.

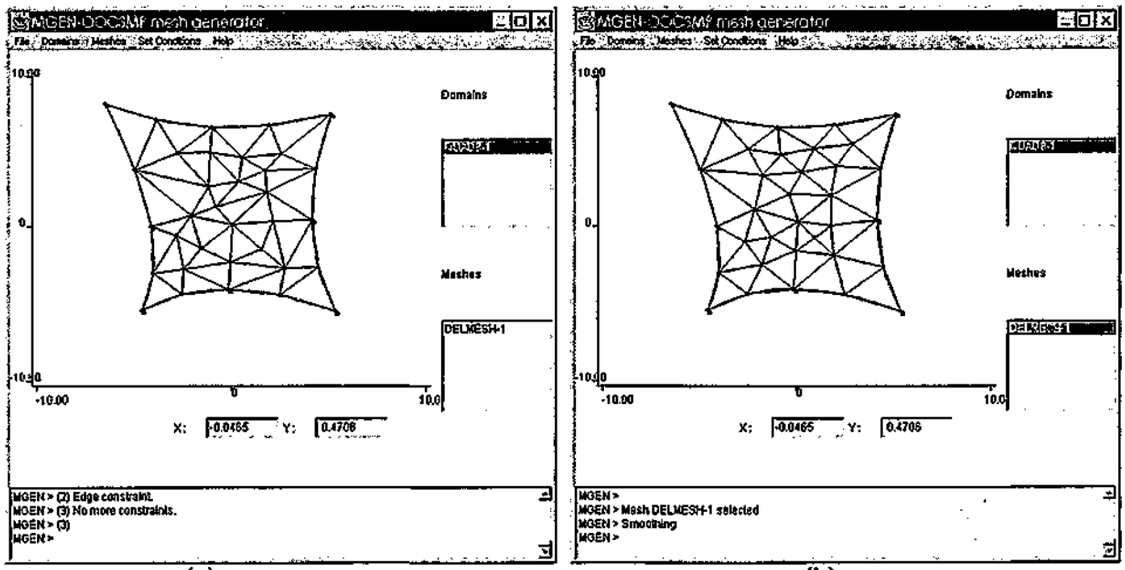


(a) (b)

Figura V.42a,b: (a) Malla generada correctamente, (b) Malla generada incorrectamente

V.7.2.3.5 Acción Meshes: Smooth

Esta acción permite suavizar (mediante el suavizado Laplaciano) una malla generada mediante interpolación o Delaunay. Está accesible desde el menú Meshes, o mediante el comando *SMOOTH*. El suavizado Laplaciano consiste en acercar progresivamente cada nodo a la media de las coordenadas de todos sus vecinos, siempre que la nueva posición del nodo caiga dentro del dominio a discretizar. Las dos siguientes figuras muestra el resultado de discretizar el mismo dominio, pero la figura V.44b ha sido suavizada.



(a) (b)

Figura V.43a,b: (a)Triangulación de un *Quad8*, (b) Triangulación y suavizado.

V.7.2.4 Menú "Set Conditions"

V.7.2.4.1 Acción Set Conditions: Boundary

Mediante esta acción es posible imponer condiciones de contorno a un dominio o malla. Está accesible desde el menú Set Conditions, o mediante la sentencia *BOUNDARY* en la línea de comandos.

Una vez seleccionado el dominio en el que se quieren imponer las condiciones, se ha de indicar si se quieren imponer sobre lados, vértices o nodos de la malla que se va a generar.

Vértices y lados se numeran en sentido antihorario, empezando por el lado inferior izquierdo. Los nodos de la malla se numeran de izquierda a derecha y de abajo a arriba. A continuación, hay que indicar los grados de libertad, elegir entre condiciones esenciales, naturales o periódicas, e introducir la expresión *OOCSMP*, en el caso de que estemos introduciendo condiciones naturales o esencales. Si se está ejecutando *MGEN* como parte de la simulación, se puede elegir la expresión *OOCSMP* de entre las funciones dinámicas (ver sección V.2)

V.7.2.4.2 Acción Set Conditions: Initial

Mediante esta acción es posible imponer condiciones iniciales (derivativas o no) a un dominio o malla. Está accesible desde el menú Set Conditions, o mediante la sentencia *INITIAL* en la línea de comandos.

Una vez seleccionado el dominio en el que se quieren imponer las condiciones, se han de indicar los grados de libertad, elegir entre condiciones derivaticas o no, e introducir la expresión *OOCSMP*. Si se está ejecutando *MGEN* como parte de la simulación, se puede elegir la expresión *OOCSMP* de entre las funciones dinámicas (ver sección V.2, construcción de dominios).

V.7.2.5 Menú "Help"

V.7.2.5.1 Acción Help: About

Esta acción muestra un cuadro de diálogo con diversa información sobre *MGEN*. Está accesible desde el menú Help, o mediante la sentencia *ABOUT* de la línea de comandos.

V.7.2.5.1 Acción Help: Meshing with *MGEN*

Muestra información detallada sobre los comandos de *MGEN*. Está accesible desde el menú Help, o mediante la sentencia *HELP* de la línea de comandos.

■ V.7.3 Uso de *MGEN* dentro de una simulación

Es posible incorporar *MGEN* dentro de una simulación generada por *C-OOL* (Ver sección V.2). De esta forma se puede crear y modificar dinámicamente el dominio, la malla y asignar las condiciones iniciales y de contorno durante la simulación.

Las diferencias entre la funcionalidad de la herramienta como generador de código y como panel en la simulación son :

- Dentro de una simulación *MGEN* no puede leer una descripción de problema de fichero, ni escribirla (las opciones desaparecen del menú *File*).
- Dentro de una simulación, se pueden usar las funciones dinámicas definidas en el modelo como condiciones iniciales y/o de contorno.
- Dentro de una simulación, se pueden asignar y/o desasignar las *PDEs* definidas en el modelo a las mallas que se diseñen con la herramienta.

La siguiente figura representa una ventana típica de *MGEN* funcionando dentro de una simulación.

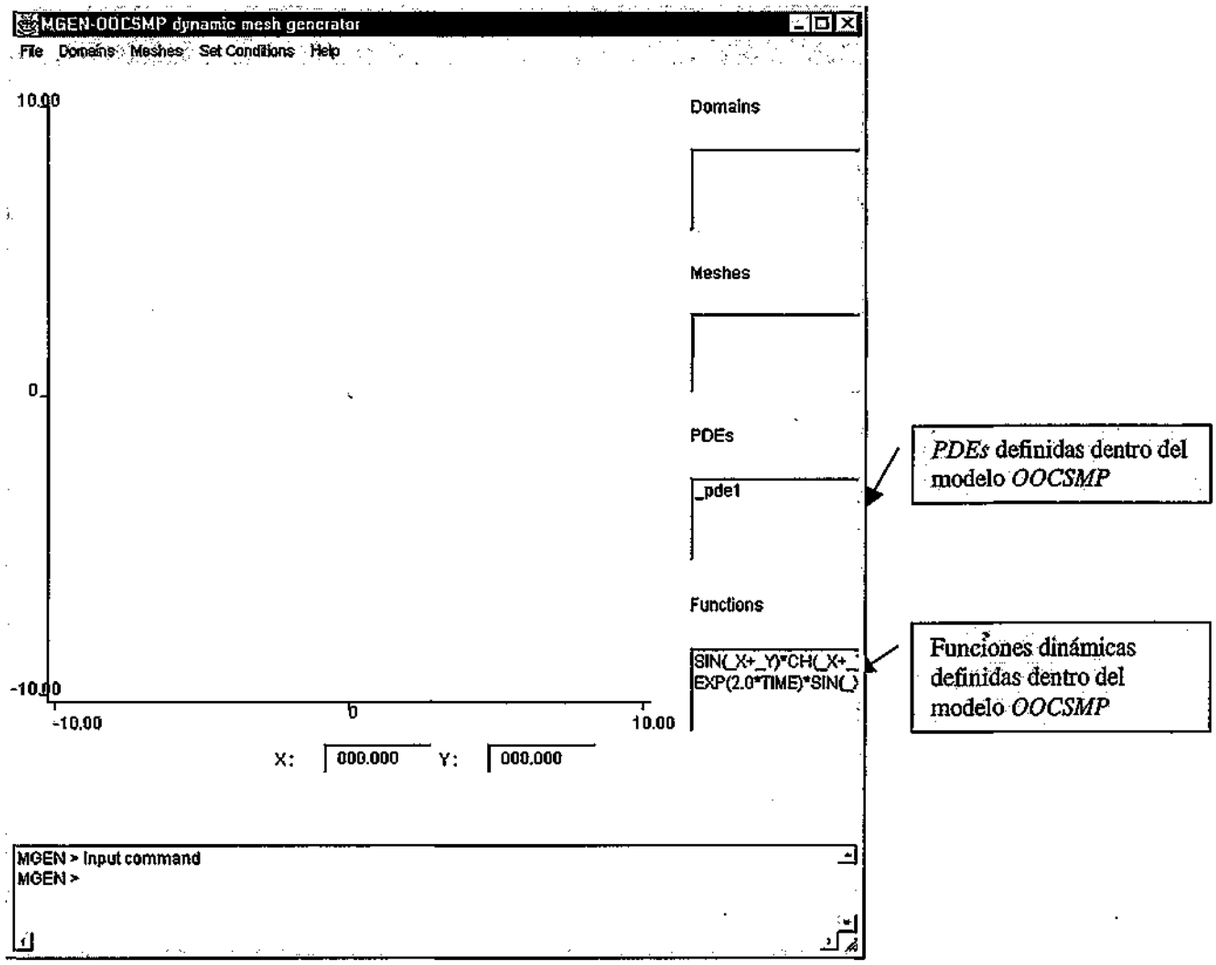


Figura V.44: Una ventana típica de *MGEN* dentro de una simulación.

Cuando se detiene la simulación para realizar los cambios pertinentes en la descripción del problema, se puede retornar a la simulación mediante la acción *"Return to simulation"* del menú *File*. Se presenta un ejemplo del uso de *MGEN* en las sección VI.2.4.2.

En el menú *Meshes* se han añadido dos nuevas opciones, que se detallan a continuación.

V.7.3.1 Acción Meshes: Set equation to Mesh.

Esta acción permite asignar una *PDE* a una malla. Se accede mediante el menú Meshes, o bien mediante la sentencia *SETPDE* en la línea de comandos. Para completar la acción, se ha de seleccionar la malla y la ecuación.

V.7.3.2 Acción Meshes: Unset equation to Mesh.

Esta acción permite desasignar una *PDE* de una malla. Se accede mediante el menú Meshes, o bien mediante la sentencia *UNSETPDE* en la línea de comandos. Para completar la acción, se ha de seleccionar la malla, y se desasignarán todas las ecuaciones que tengan asignadas.

VI. Salidas gráficas y multimedia

En el presente capítulo, se van a tratar las distintas salidas gráficas y las extensiones multimedia que han sido incorporados al lenguaje *OOCSMP*.

En la siguiente sección, VI.1, introducción, se presenta un esquema del formato de la interfaz de usuario que el compilador genera para el lenguaje Java, ya que es el lenguaje que vamos a usar para la construcción de los cursos, y es con la opción de compilación a Java con la única que van a funcionar la mayoría de las salidas gráficas. Otros detalles de la interfaz Java, así como las interfaces C++/*DOS* y C++/Amulet se detallan en el capítulo IX.

En la sección VI.2, se detalla cada tipo de salida gráfica, junto con ejemplos de su uso. En la sección VI.3, se presentan las primitivas multimedia, también con ejemplos.

VI.1 Introducción

La visualización de los datos de simulación en formato flexible y adecuado al contenido de esos datos, debe ser una característica básica de una herramienta de simulación. El usuario debe poder ver los datos en distintos formatos sin demasiado esfuerzo.

Nuestro compilador, *C-OOL*, genera código Java, *C++/DOS*, o C++/Amulet con los modelos *OOCSMP*. La mayoría de las salidas gráficas sólo están disponibles en el caso de que se genere Java, ya que estos formatos gráficos nos eran de mayor utilidad en los cursos de Internet. El código C++ generado se usa principalmente para incluirlo en alguna aplicación que necesite mayor rendimiento.

La interfaz de usuario que *C-OOL* genera para el caso de que se elija Java como lenguaje objeto se divide en varias partes:

- Un panel principal con capacidad máxima para 9 representaciones gráficas, o elementos multimedia. El panel es una malla de tamaño tres por tres, pero si no se rellenan todos los huecos, la malla toma el tamaño más pequeño posible automáticamente. Situados dentro de este panel, se pueden encontrar:
  - Dos o tres objetos gráficos (dependiendo de las salidas gráficas seleccionadas), en forma de barras de desplazamiento, que permiten ajustar el tiempo final de simulación, el tiempo actual, y el intervalo de refresco de la salida gráfica tipo *ICONICPLOT*.
  - Varios botones que permiten ver y modificar los valores de las variables de los objetos de la simulación, y las variables globales.
  - Botones que permiten añadir o borrar objetos de simulación en tiempo de ejecución.
- Hasta nueve ventanas adicionales con distintas representaciones gráficas.

En el caso de Java, se crean dos hilos distintos: uno para los cálculos del bucle de simulación, y otro para la interfaz gráfica, si bien algunas salidas gráficas también crean su propio hilo. La figura VI.1 muestra la organización típica de una interfaz de usuario generada con *C-OOL*.

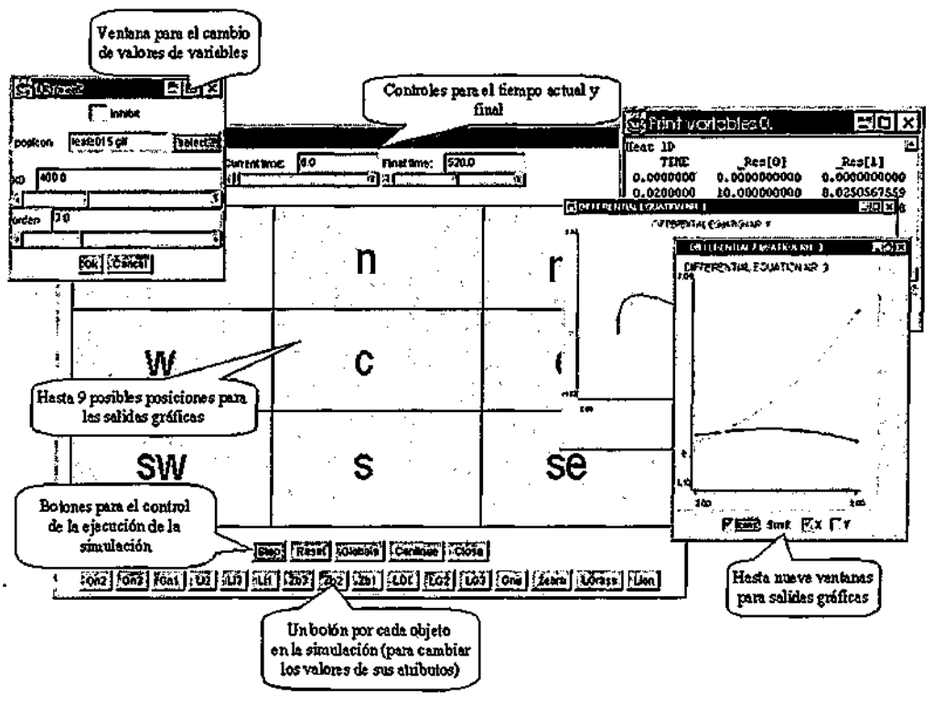


Figura VI.1: Esquema de una interfaz típica para Java.

El panel principal se divide en nueve huecos. Cada salida gráfica toma como primer parámetro los huecos que va a ocupar, separados por comas (*NW, N, NE, W, C, E, SW, S, SE*), o bien *WINDOW* si la salida debe aparecer como una ventana separada. Los huecos que ocupa un panel han de ser contiguos. Los tipos de salida gráfica y elementos multimedia disponibles en *OOCSMP* son los siguientes:

- Listados de valores de variables (*PRINT*).
- Gráficos bidimensionales, que se usan para representar funciones unidimensionales. Este gráfico se puede animar, en cuyo caso, se puede asignar un icono a cada variable dependiente. El nombre del fichero con el icono se puede especificar en el atributo *ICON* del objeto (ver sección IV.2.2.1) que se dibuja, o directamente en la sentencia *PLOT*.
- Gráficos bidimensionales para vectores (*PLOT2D*). Esta salida gráfica permite mostrar de forma animada el valor que toma un vector o la solución de una *PDE* en una dimensión (a intervalos de tiempo de `PLdelta`).
- Gráficos en tres dimensiones para matrices (*PLOT3D*). Esta salida gráfica crea su propio hilo, y permite rotar la superficie (pulsando el botón '*rotate*'), escalar la superficie (*shift+click* del ratón+mover el ratón), desplazar la superficie (*Ctrl+click* del ratón+mover el ratón), y cambiar el ángulo de visualización (*click* del ratón+mover el ratón). Esta salida gráfica también puede aplicarse a mallas usadas para la resolución de una *PDE*.
- Gráficos icónicos (*ICONICPLOT*), que representan por medio de iconos la variación con respecto al tiempo de varias variables. El número de iconos visibles de la misma clase es proporcional al valor de la variable que representan. Esta salida gráfica crea su propio hilo.
- Representaciones gráficas de las ecuaciones (*CONNECTIONPLOT*). En esta salida gráfica, aparece un bloque gráfico por cada bloque *OOCSMP*. Esta representación también permite la introducción interactiva de datos de entrada en el modelo.
- Mapas de isosuperficies que pueden ser aplicados a matrices o a mallas (*ISOPLOT*).
- Gráfico con los nodos de una malla y sus condiciones iniciales (*GRIDPLOT*). En este gráfico es posible hacer zoom de diversas zonas de la malla, y cambiar la ecuación de la que estamos viendo la condición inicial.
- Panel de imágenes (*IMAGEPANEL*): Esta extensión multimedia, presenta un panel, o una ventana, donde se van mostrando varias imágenes. Cada imagen se presenta cuando se cumple cierta condición asociada en la simulación. La condición puede ser cualquier expresión lógica *OOCSMP*.
- Panel de vídeos (*VIDEOPANEL*): Igual formato que el anterior, pero mostrándose vídeos en lugar de imágenes. Para la realización de este tipo de panel, se ha utilizado el Java Media Framework v2.0 [JMF99], aceptándose secuencias de vídeo de tipo *mov* y *avi*.
- Audio (*AUDIOPANEL*): Igual estructura que los dos anteriores, pero con audio. Este elemento multimedia no tiene un objeto gráfico asociado, acepta secuencias de audio de tipo *wav*, y también se ha construido usando el Java Media Framework.
- Panel de Texto (*TEXTPANEL*): Igual que los tres anteriores, pero con texto.

En el caso de que se genere C++/*DOS* ó Amulet, sólo se puede elegir una forma de representación en el panel principal, además de un listado de valores de variables. La única salida gráfica disponible para estos dos casos es el gráfico bidimensional.

VI.2 Uso de las salidas gráficas

En esta sección se presentarán ejemplos de uso de las distintas salidas gráficas, que combinan varias de ellas en un mismo problema.

VI.2.1 Uso de las instrucciones *ICONICPLOT* y *PLOT*

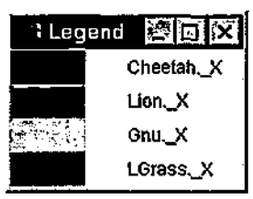
Sunpongase que se quiere simular el sistema ecológico de la sabana africana[Alfo98c] como parte de un curso para Internet. Una vez que se ha programado el modelo, se ha de decidir cuáles van a ser las salidas gráficas para el applet Java. Para este problema, podría ser conveniente mostrar dos gráficos:

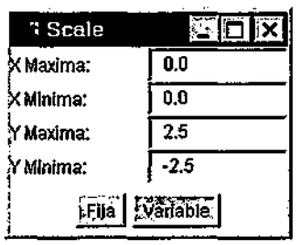
- Un gráfico 2D (*PLOT*) que muestre la evolución de las poblaciones de las distintas especies del ecosistema. La sintaxis *OOCSMP* de la instrucción *PLOT* es:

```
PLOT  [[posicion],] [[LG=position],] [[SC=position],]
      [[MACHINE= etiq-maquina],]
      [maxValX, maxValY [,minValX, minValY,]]
      (dependent-var [,"filename.gif"],)* independent-var
```

Sintaxis VI.1: Sintaxis de la instrucción *PLOT*.

LG significa '*LeGend*' y muestra un panel con el color asociado a cada variable que se dibuja. *SC* significa '*SCale*', y muestra un panel que nos permite cambiar la escala del gráfico. Las dos siguientes figuras muestran el aspecto de los paneles de leyenda y escala cuando se insertan en la simulación en ventanas separadas.





Figuras VI.2 y 3: Paneles de leyenda y escala.

- *MACHINE= etiq-maquina* es una opción válida sólo si estamos generando código distribuido, y será detallada en el capítulo VII.

- *maxValX* y *maxValY* son los valores máximos para los ejes *X* e *Y*. Si no se especifica ninguno y la variable independiente es *TIME*, *maxValX* toma el valor del tiempo final de simulación (variable *FINTIM*).

- *minValX* y *minValY* son los valores mínimos para los ejes *X* e *Y*. Si no se especifica ninguno de los dos, pero los máximos sí, toman el valor de *-maxValX* y *-maxValY*, respectivamente.

- *dependent-var* son las variables dependientes que se van a dibujar. Aquí también podemos especificar *<nombre-clase>.<nombre-variable>*, que significa que se va a dibujar el atributo *<nombre-variable>* de todos los objetos que pertenezcan a las clase *<nombre-clase>*. También es posible la sintaxis *<nombre-colección>.<nombre-variable>* con semántica similar (el atributo *<nombre-variable>* de todos los objetos que pertenezcan a la colección). Si se especifica una sentencia *PLOT* dentro de una definición de una clase, entonces se dibujarán los valores de las variables especificadas para todos los objetos de dicha clase.

- Finalmente, *independent-var* es la variable independiente del problema.

Las variables se dibujan en cada intervalo de tiempo múltiplo de *PLdelta*.

- Un gráfico icónico que muestre esta misma información de forma más visual (un icono por cada individuo de cada especie). La instrucción *OOCSMP* que nos muestra dicho gráfico es *ICONICPLOT*, que tiene la siguiente sintaxis:

```
ICONICPLOT [(posicion),] [(MACHINE=etiq-maquina),]
           (var [,"filename.gif"])*
```

Sintaxis VI.2: Sintaxis de la instrucción *ICONICPLOT*.

Donde *posicion* es la posición donde se colocará el gráfico, *var* es cada una de las variables que se van a dibujar, y *"filename.gif"* es el nombre del fichero que contiene el icono que se dibujará, asociado a la variable *var*.

MACHINE="nombre maquina" es una opción válida sólo si estamos generando código distribuido, y será detallada en el capítulo VII.

Supóngase también que se quiere mostrar sólo unas cuantas cadenas tróficas a la vez, debido a que hay muchas. Esto lo podemos hacer en *OOCSMP* con la instrucción '\' (ver sección IV.6.3). Después de esta instrucción, se pueden añadir nuevas sentencias para salidas gráficas. Las nuevas salidas gráficas reemplazan a las que estuvieran en el lugar indicado. Además, la instrucción '\' también genera un botón que permite cambiar entre las representaciones gráficas. El listado VI.1 muestra un esquema del modelo *OOCSMP* y la figura VI.4 su ejecución.

En el ejemplo de la figura VI.4, las instrucciones '\' crean tres botones diferentes, uno que muestra la simulación principal, los otros dos que muestran las cadenas alimenticias 2 y 3. Las ventanas de escala y leyenda se han suprimido mediante opciones de compilación. Los botones para cambiar los parámetros de las especies también se han suprimido.

Este applet se encuentra en Internet en: *http://www.ii.uam.es/~epulido/ecology/eco6.htm*

```
TITLE Savanna
CLASS Species
(
  NAME name
  ICON icname
  * behaviour of a generic specie
  ...
)
* Definition of all the species
Species Lion("Lion","lion002.gif",...)
...
Species EcoSystem := Lion,...
* An array containing all the species
DYNAMIC * The main simulation loop
...
PLOT        [C],Lion.X, ..., TIME
ICONICPLOT [S],Lion.X,...
\
TITLE Chain 2
PLOT [C], Lion.X, Giraffe.X, ..., TIME
\
TITLE Chain 3
PLOT [C], Cheetah.X, ..., TIME
```

Listado VI.1: Esquema del modelo de la sabana africana.

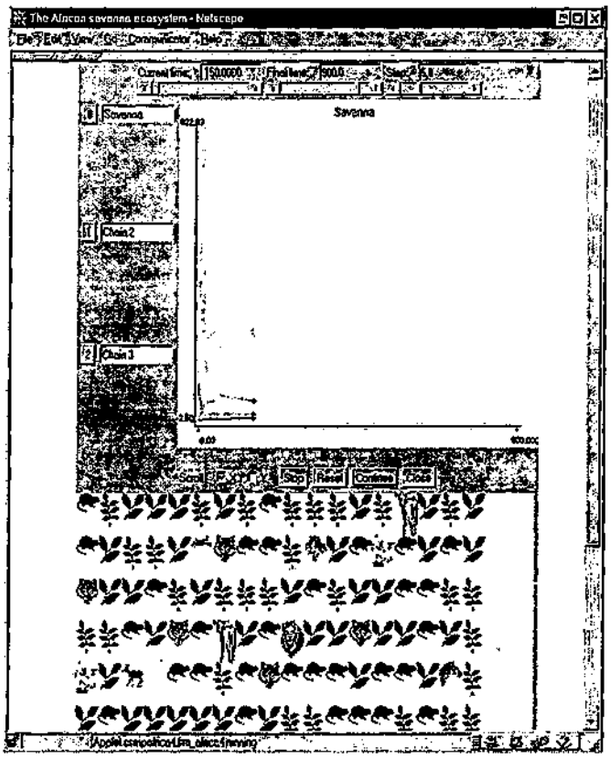


Figura VI.4: *Applet* del modelo de la sabana.

■ VI.2.2 Uso de la instrucción *CONNECTIONPLOT*

Supongamos que queremos simular el comportamiento de un circuito electrónico [Alfo98d]. Queremos mostrar el circuito como salida gráfica, y poder cambiar las entradas gráficamente, ver el valor de las puertas lógicas intermedias, y las salidas. Esto se puede llevar a cabo por medio de la instrucción *CONNECTIONPLOT*.

Esta salida gráfica es una representación gráfica de las ecuaciones del modelo, es decir, genera un bloque gráfico por cada bloque que aparezca en la ecuación, conecta las entradas y salidas, y genera controles especiales para las entradas y salidas del modelo. En nuestro caso, la representación gráfica que obtenemos es un circuito electrónico. El usuario puede cambiar las entradas del circuito pinchando en los controles de entrada (como son binarios, sólo cambian de cero a uno al ser pinchados). También es posible cambiar las entradas del modelo aunque éstas sean escalares, vectores o matrices. La sintaxis de la instrucción *CONNECTIONPLOT* se muestra a continuación:

```
CONNECTIONPLOT [(posicion)]
               [, (MACHINE= etiq-maquina)]
               [, (objeto.metodo | bloque)]
               [, LED7]
```

Sintaxis VI.3: Sintaxis de la instrucción *CONNECTIONPLOT*.

Donde:

- Como en apartados anteriores, `posicion` especifica la posición en la que se quiere colocar la salida gráfica.

- *MACHINE= etiq-maquina* es una opción válida sólo si estamos generando código distribuido, y será detallada en el capítulo VII.

- *Objeto.metodo* especifica que la salida será una representación gráfica de las ecuaciones contenidas en dicho método y objeto.

- *Bloque* especifica lo mismo, pero para un bloque definido por el usuario.

- *LED7* indica que queremos un led de siete segmentos para la salida gráfica.

Los siguientes listados muestran la construcción de un sumador de cuatro bits, a partir de cuatro sumadores de un bit.

```
CLASS adder1
{
        NAME name
        ICON icName:="add1.gif"
        DYNAMIC A,B,C
            xor1 := EOR ( A,B )
            and1 := AND ( A,B )
            and2 := AND ( xor1, C )
            CARRY:= IOR ( and1, and2 )
            OUT  := EOR ( xor1, C )
}
```

Listado VI.2: Una clase con un sumador de un bit (fichero *circ/ADDER1.CSM*).

```
INCLUDE "circ\ADDER1.CSM"
TITLE 4 bit adder using 1 bit adders
ADDER1 a1("a1")
ADDER1 a2("a2")
ADDER1 a3("a3")
ADDER1 a4("a4")
ADDER1 sumadores := a1,a2,a3,a4
DATA OUT[4]
DATA X0:=0,X1:=0,X2:=0,X3:=0,Y0:=0,
Y1:=0,Y2:=0,Y3:=0,C:=0
DYNAMIC
    a1.STEP ( X0 , Y0 , C )
    a2.STEP ( X1 , Y1 , a1.CARRY )
    a3.STEP ( X2 , Y2 , a2.CARRY )
    a4.STEP ( X3 , Y3 , a3.CARRY )
    CARRY := a4.CARRY
    OUT   := sumadores.OUT
TIMER FINTIM:=1, delta:=1
CONNECTIONPLOT
```

Listado VI.3: un sumador de 4 bits.

El resultado de la compilación del listado VI.3 se muestra en la figura VI.5 y se puede encontrar en la dirección Internet: *www.ii.uam.es/~epulido/circ/modules.htm*

Como se puede observar, la representación gráfica de los sumadores de 1 bit, se coge del atributo *ICON* de la clase *adder1*. Para todos los objetos (*a1..a4*) es el fichero *add1.gif*. Las salidas del modelo VI.3 son el vector *OUT* (la suma de los dos números) y la variable *CARRY* (el acarreo), las entradas son *X0...X3* (primer número), *Y0...Y3* (segundo número) y *C* (el acarreo).

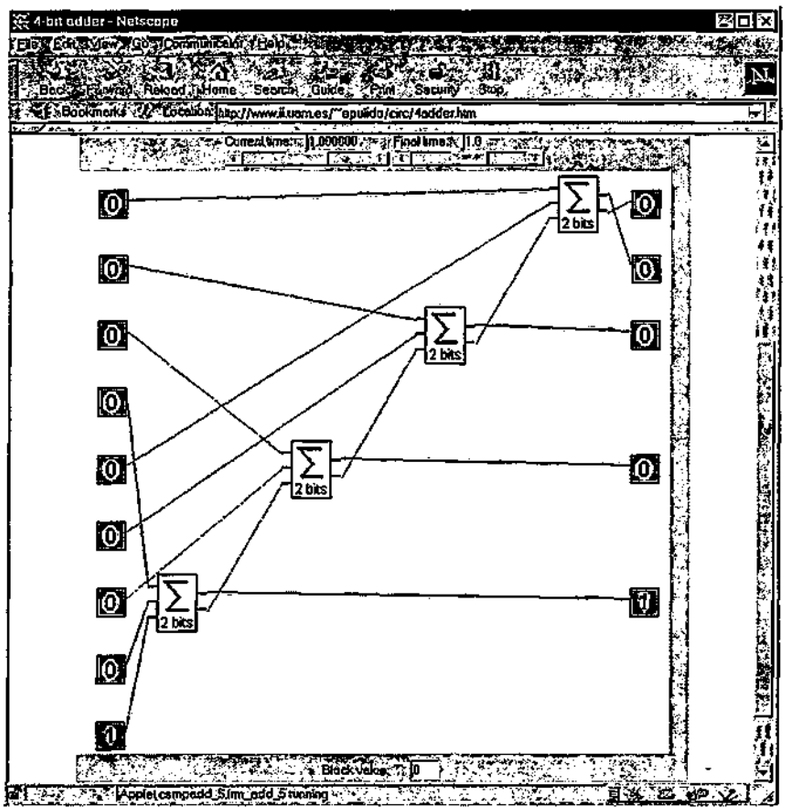


Figura VI.5: Un sumador de 4 bits.

■ VI.2.3 Uso de los gráficos 3D y de los gráficos para vectores

Supongamos que queremos simular el calentamiento de una barra [deLa99], [Alfo99c]. La ecuación que gobierna este fenómeno es la ecuación del calor [Redd93], que es:

$$\rho cA\frac{dT}{dt} - \frac{d}{dx}\left(kA\frac{dT}{dx}\right) - Aq = 0$$

Ecuación VI.1: Ecuación del calor en una dimensión.

Donde ρ es la densidad, c es el calor específico del material, A es el área de la sección del material, T es la temperatura, k es la conductividad térmica, t es el tiempo, y q la energía calorífica generada por unidad de volumen.

Podríamos usar un gráfico para vectores para ver la temperatura en cada intervalo de tiempo. Este vector se dibujará cuando el tiempo de la simulación sea múltiplo de `PLdelta`. La sintaxis de esta instrucción es:

```
PLOT2D      [[posicion],] [[MACHINE=etiq-maquina],]
            [maxX,maxY[,minX, minY]],vector
```

Sintaxis VI.3: Sintaxis de la instrucción *PLOT2D*.

Donde los parámetros tienen el mismo significado que para la instrucción *PLOT*. El parámetro `vector`, puede ser bien un vector, o una malla unidimensional. En el primer caso, se ha de especificar al menos la escala en el eje X, asignándose a cada nodo del vector una coordenada $x_i = minX+i*(maxX-minX)/(n-1)$, donde n es el número de elementos del vector. El contenido de esta salida gráfica es animado, refrescándose a intervalos de `PLdelta`.

También es posible mostrar un gráfico tridimensional, con ejes: la dimensión de la barra que se calienta (eje *X*), el tiempo (eje *Y*), y la temperatura (eje *Z*). Para ello podemos usar la instrucción *PLOT3D*, que tiene como sintaxis:

```
PLOT3D      [[posicion],] [[MACHINE= etiq-maquina],]
            [maxX,maxY[,minX, minY],] variable
```

Sintaxis VI.3: Sintaxis de la instrucción *PLOT3D*.

Donde los parámetros tienen el mismo significado que para la instrucción *PLOT*. En este caso, `variable` puede ser:

- Un vector, en cuyo caso se escoge como eje *Y* el tiempo, y se ha de especificar la escala.
- Una matriz y se ha de especificar la escala.
- Una malla bidimensional.
- Una malla unidimensional, en cuyo caso se coge como eje *Y* el tiempo.

Supongamos que la función *q* de la ecuación VI.1 depende del tiempo. Se podría visualizar en un gráfico bidimensional, y también obtener otra información, por ejemplo el calor total de la barra. Vamos a poner estas dos últimas funciones en dos gráficos distintos, ya que las escalas de *q(t)* y del calor podrían ser bastante distintas.

Por último, queremos que aparezca un listado con el valor del calor en cada punto de la malla en distintos intervalos de tiempo. Esto lo podemos hacer con la instrucción *PRINT*, que tiene como sintaxis:

```
PRINT [[posicion],] [[MACHINE= etiq-maquina],] var1 (,var2)*
```

Sintaxis VI.4: Sintaxis de la instrucción *PRINT*.

Donde los parámetros tienen el mismo significado que en salidas gráficas anteriores. `var1`, `var2`, etc., son las variables que queremos visualizar. Pueden ser variables simples, vectores o matrices. También se aceptan expresiones del tipo:

- *<objeto>.<atributo>*, en cuyo caso se imprimirá el valor de *<atributo>* del objeto *<objeto>*.
- *<clase>.<atributo>*, en cuyo caso se imprimirá el valor de *<atributo>* de cada objeto de la clase *<clase>*.
- *<colección-objetos>.<atributo>*, en cuyo caso se imprimirá el valor de *<atributo>* de cada objeto de la colección de objetos.

Al igual que con la instrucción *PLOT*, si se especifica una sentencia *PRINT* dentro de una definición de una clase, entonces se imprimirán los valores de las variables especificadas para todos los objetos de dicha clase.

Se realiza una impresión de estas variables cuando el tiempo de la simulación es múltiplo de `PRdelta`. El siguiente listado muestra el código *OOCSMP* necesario para resolver el problema, y la figura VI.6 el *applet* generado.

```
TITLE Heat 1D
DATA K:=4.0
DOMAIN bar1d:=BAR(0.0,4.0,
       INITIAL(0.0),
       ESSENTIAL(EDGE(1),10.0))
MESH Res := ISOPARAMETRIC(bar1d, LINE2, ELEMENTS(40) )
PDE H1d(0,0,1,1,-K,0,0,0,0,0,0,0,0,0,
    -SIN(TIME*3.1415/FINTIM)*(TIME+2.0),
    EXPLICIT, TScheme(FORWARD) )
Res.setPDE(H1d)
Res.STEP()
AUX := SIN(TIME*3.1415/FINTIM)* (TIME+2.0)
SUMA:= +Res.VALUE()
TIMER delta:=0.0005, FINTIM:= 0.04, PLdelta:=0.001, PRdelta:=0.02
```

```
PLOT3D [C], Res
PLOT2D [E], Res
PLOT   [S], SUMA, TIME
PLOT   [SE],AUX, TIME
PRINT  [WINDOW], Res
```

Listado VI.4: Resolución de la ecuación del calor en una barra.

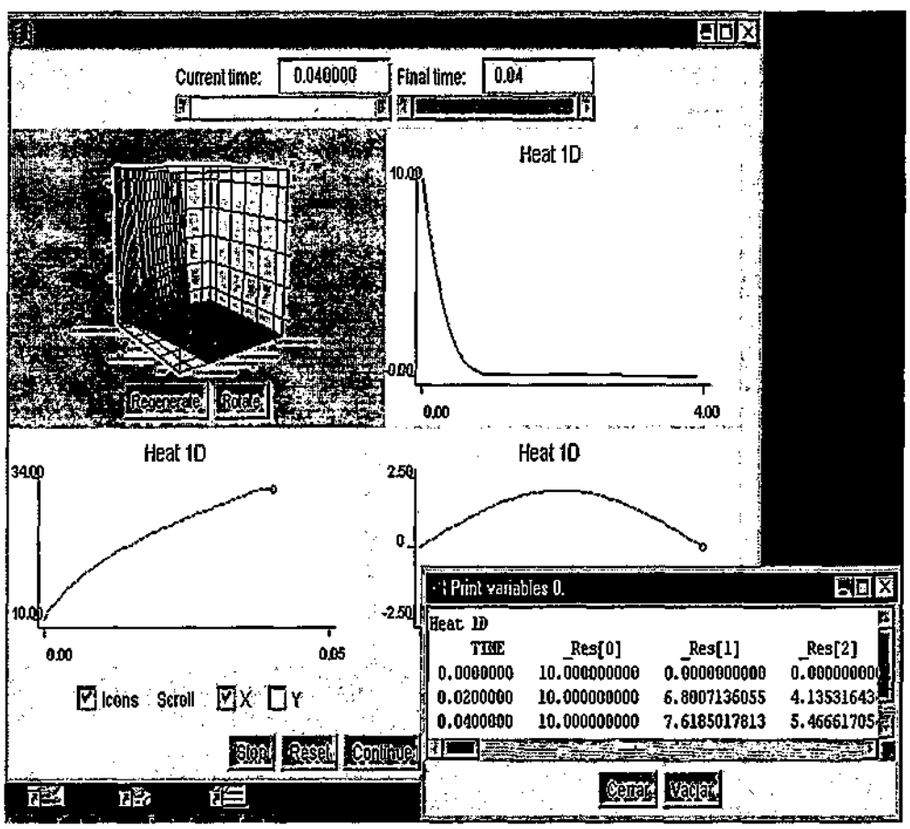


Figura VI.6 : Resolución de la ecuación del calor en 1-D.

■ VI.2.4 Uso de gráficos de mallas, mapas de isosuperficies y *MGEN*

Supóngase que se quiere resolver la siguiente ecuación en dos dimensiones [Stri89]:

$$\frac{dU}{dt}+\frac{dU}{dxx}+\frac{dU}{dxy}+\frac{dU}{dyy}=0$$

Ecuación VI.2

Cuya solución exacta viene dada por la expresión: $e^{2t}\sin(x+y)\cosh(x+y)$.

Podríamos usar dos mapas de isosuperficies:

- Uno para la solución calculada (una malla).
- Otro para la solución exacta (una matriz).

La sintaxis de los mapas de superficies es la siguiente:

```
ISOPLOT     [[posicion],] [[SMOOTH],]
            [minX, minY, maxX, maxY,]
            [ [MACHINE= etiqueta-maquina],]
            <mesh-1> (,<mesh-n>)*
```

Sintaxis VI.5: Sintaxis de la instrucción *ISOPLOT*.

- Donde los parámetros *[POSITION]* y *[MACHINE= ...]* tienen igual significado que para bloques anteriores.
- La opción *[SMOOTH]* sólo es aplicable cuando estamos dibujando un mapa de isosuperficies de una matriz, e indica si queremos que al dibujar cada punto se haga un promedio con los vecinos o no. Es decir, si incluimos esta opción, se dibuja la matriz como se dibujaría una malla rectangular con elementos cuadrados de cuatro nodos. El valor de cada elemento de la malla sería el promedio del valor de los cuatro nodos del elemento.
- *minX, minY, maxX* y *maxY* son las coordenadas máximas y mínimas de los ejes X e Y, para el caso de que estemos representando una matriz. En las mallas no haría falta, ya que la información de las coordenadas está incluida en la definición de la malla.
- *<mesh-1>* es el identificador de una malla bidimensional (simple o compuesta). Se pueden especificar una o varias mallas. Este panel se acompaña de una leyenda con los rangos de valores (5) asociados a los colores del mapa..

Para el presente problema, también podríamos usar un gráfico con los nodos de la malla con las condiciones iniciales. Este se puede llevar a cabo con la instrucción *GRIDPLOT*, que tiene la siguiente sintaxis:

```
GRIDPLOT    [[posicion],] [[MACHINE=etiq-maquina],]
            <mesh-1> (,<mesh-n>)*
```

Sintaxis VI.5: Sintaxis de la instrucción *GRIDPLOT*.

Con el mismo significado de los parámetros que para bloques anteriores. En este panel también se pueden visualizar las condiciones iniciales. Los nodos de la malla aparecen coloreados dependiendo del valor de su condición inicial. Este panel se acompaña de una leyenda con los rangos de valores (5) asociados a los colores de los nodos.

Vamos a resolver este problema de dos formas, la primera, resolviendo la ecuación sobre un rectángulo, al que ponemos como condiciones iniciales y de contorno la solución exacta. La segunda, declarando nuestro dominio con *MGEN* durante la simulación, poniendo condiciones iniciales y de contorno dentro de *MGEN*.

VI.2.4.1 Resolución sin usar *MGEN*

El listado VI.5 muestra el modelo *OOCSMP* del problema sin el uso de *MGEN*. La figura VI.5 el resultado.

```
DATA exacta[75;75]
DOMAIN qd := QUADRILATERAL(-1, -1, 1, -1, 1, 1, -1, 1,
  INITIAL(SIN(X+Y)*CH(x+Y)),
  ESSENTIAL(EDGE(1:4),
  EXP(2.0*TIME)*SIN(x+y)*CH(x+y)))

MESH m := ISOPARAMETRIC( qd, QUADRILAT4, ELEMENTS(75,75) )
PDE pde1 ( 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, DUFORTE )
m.setPDE(pde1)
DYNAMIC
exacta[ROW;COL] := EXP(2.0*TIME)*SIN((-1+COL*0.027)+(-
1+ROW*0.027))*
CH((-1+COL*0.027)+(-1+ROW*0.027))
m.STEP()
TIMER FINTIM:=1.0, delta:=0.05, PLdelta:=0.1
```

```
ISOPLOT  [C], m
ISOPLOT  [S], [SMOOTH], 1, 1, -1, -1, exacta
GRIDPLOT [E], m
```

Listado VI.5: Resolución del problema sin usar *MGEN*.

Como se puede observar, se ha declarado una matriz en la que se va a ir guardando la solución exacta en cada paso de tiempo. Para la malla, se ha elegido un generador de mallas isoparamétrico, con 75 elementos en las direcciones *X* e *Y*. En el mapa de isosuperficies de la solución exacta, se ha tenido que especificar la escala, ya que es una matriz.

Este applet se puede encontrar en:
http://www.ii.uam.es/~jlara/investigacion/ecomm/pde1.html

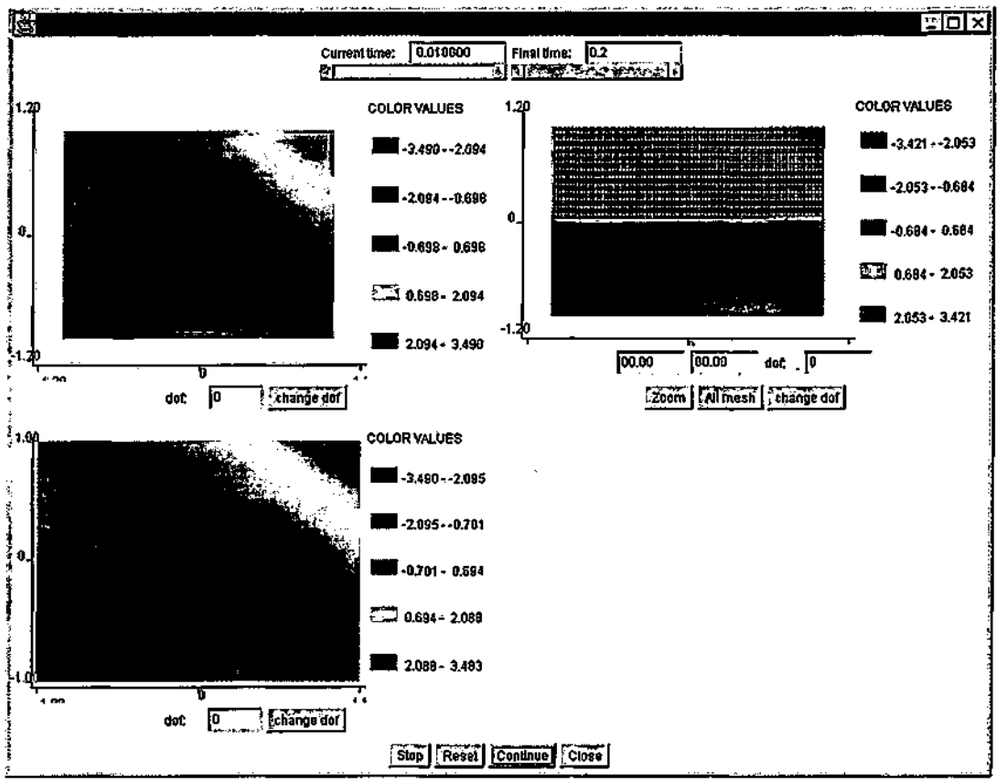


Figura VI.7: Resolución del problema del listado VI.5.

VI.2.4.2 Resolución con *MGEN*

Supongamos que queremos solucionar ahora el mismo problema, pero esta vez usando *MGEN* para el diseño del dominio (que esta vez no será un rectángulo), discretización del mismo y la imposición de las condiciones iniciales y de contorno. El listado VI.6 nos muestra el modelo una vez añadido *MGEN*. Como se puede observar sólo hemos tenido que cambiar dos líneas (la de declaración del dominio y su discretización). En la declaración del dominio, hemos indicado qué funciones queremos usar como condiciones para el problema. Además hemos suprimido la instrucción que asignaba la *PDE* a la malla, ya que esto lo haremos usando *MGEN*, en tiempo de ejecución.

```
DATA exact[75;75]
DOMAIN qd := MGEN( DYNAMIC( SIN(X+Y)*CH(X+Y) )
                  ,DYNAMIC(EXP(2.0*TIME)*SIN(X+Y)*CH(X+Y) ) )

MESH m := MGEN()
PDE pde1 ( 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, IMPLICIT )
DYNAMIC
```

```
 exact[ROW;COL] := EXP(2.0*TIME)*SIN((-1+COL*0.027)+(-1+ROW*0.027))*
                   CH((-1+COL*0.027)+(-1+ROW*0.027))
 m.STEP()
TIMER FINTIM:=0.2, delta:=0.001,PLdelta:=0.01
ISOPLOT     [C], m
ISOPLOT     [S], [SMOOTH], 1, 1, -1, -1, exact
GRIDPLOT    [E], m
```

Listado VI.6: Resolución del problema usando *MGEN*.

La figura VI.8 muestra el resultado de la ejecución del problema del listado VI.6. Lo primero que hemos hecho ha sido diseñar el dominio, en el presente caso un cuadrilátero de lados curvos, que hemos discretizado con un generador de mallas isoparamétrico. Se han impuesto las condiciones de contorno en cada uno de los ocho lados, y la condición inicial. Así mismo se ha asignado la *PDE* a la malla generada. Una vez hecho esto, se ha vuelto de *MGEN* al panel principal de la simulación y se ha empezado a simular.

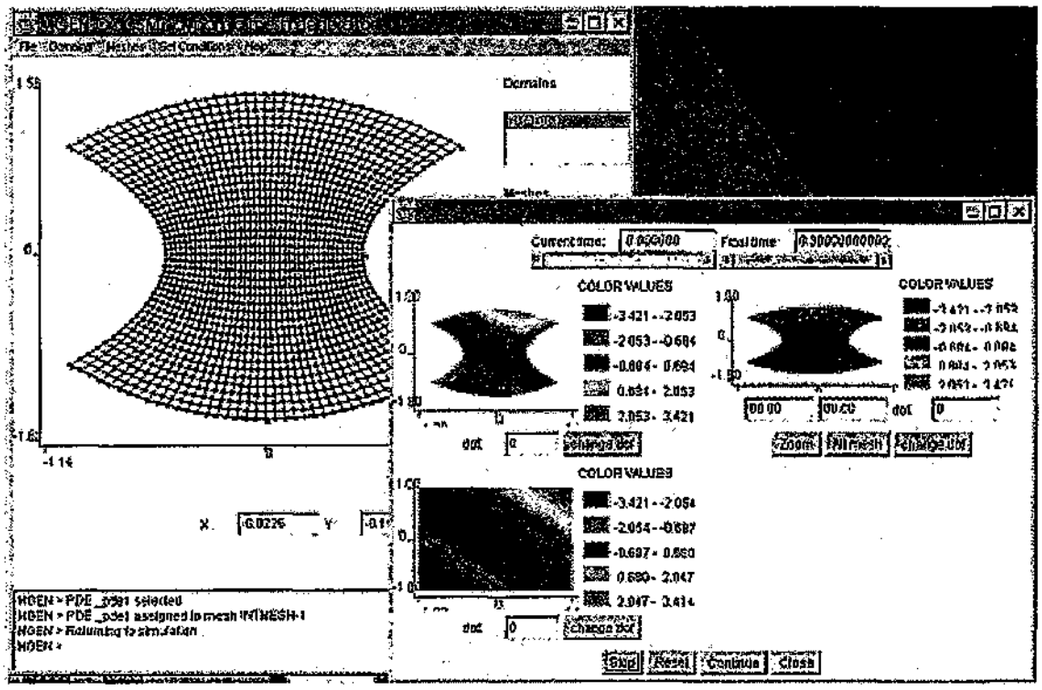


Figura VI.8: Resultado de la ejecución con *MGEN*.

VI.3 Las primitivas multimedia

Hemos añadido varias extensiones multimedia (imágenes, vídeos, audio y texto) al lenguaje *OOCSMP* [Alfo99f]. Cada una de ellas se detalla a continuación:

- Panel de imágenes: Esta extensión presenta un panel, o una ventana, en el que se van mostrando varias imágenes. Cada imagen se presenta cuando se cumple cierta condición asociada en la simulación. Dichas condiciones pueden ser cualquier expresión lógica *OOCSMP*. La sintaxis *OOCSMP* es la siguiente:

```
IMAGEPANEL [[posicion],] [[MACHINE= etiq-maquina],]
           ( START( <OOCSMPexpresion>), "file", )*
           START( <OOCSMPexpresion>), "file"
           [DEFAULT, "file"]
```

Sintaxis VI.6: Sintaxis de la instrucción *IMAGEPANEL*.

Donde `position` puede ser una de las nueve posibles posiciones (*N, S, E, W, NE, NW, SE, SW*) dentro del panel principal, una lista de ellas, separadas por comas, si ocupa más de una posición, o bien *WINDOW* y se muestra en una ventana aparte. Es decir, cuando se cumple la condición *<OOCSMPexpresion>* en la simulación, se presenta la imagen contenida en "*file*". Se pueden especificar tantos pares de condiciones e imágenes como se quieran, así como tantos paneles de imágenes como sea necesario. Si se especifica *DEFAULT*, se muestra el fichero asociado cuando no se cumple ninguna de las condiciones anteriores. El tipo de imágenes que se acepta es `gif` y `jpeg`.

- Panel de vídeos (*VIDEOPANEL*): Igual formato que el anterior, pero mostrándose vídeos en lugar de imágenes. Para la realización de este tipo de panel, se ha utilizado el Java Media Framework v2.0 [JMF99], aceptándose secuencias de vídeo de tipo `mov` y `avi`. Los vídeos se presentan sin controles de marcha hacia atrás, parada, etc., para que el alumno no pueda cambiar la sincronización planeada por el constructor del curso.

- Audio (*AUDIOPANEL*): Igual estructura que los dos anteriores, pero con audio. Además este elemento multimedia no tiene un objeto gráfico asociado. Este elemento acepta secuencias de audio de tipo *wav*, y también se ha construido usando el Java Media Framework.

- Panel de Texto (*TEXTPANEL*): Igual que los anteriores, pero con texto.

Hemos establecido un procedimiento para la integración de los elementos multimedia en los modelos de simulación, que queda esquematizado en la figura VI.9.

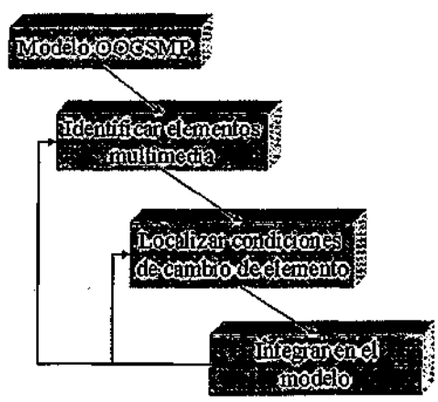


Figura VI.9: Procedimiento para la incorporación de elementos multimedia a la simulación.

El procedimiento de la figura VI.9, parte del modelo *OOCSMP*. El primer paso consiste en identificar qué elementos multimedia vamos a poner en la simulación (videos, imágenes, etc.). El siguiente paso consiste en localizar cuándo los vamos a poner, e identificar la expresión *OOCSMP* que va a provocar el cambio de un elemento del mismo tipo a otro, por ejemplo, cuándo se cambia de una imagen a la siguiente, etc. El último paso consiste en programarlo en el modelo y decidir en qué posición del panel principal lo vamos a mostrar, o si vamos a ponerlos en ventanas separadas.

A continuación se presentan varios ejemplos de uso de estas primitivas multimedia y del procedimiento asociado.

VI.3.1 Simulación de un ecosistema con elementos multimedia (imagen y texto)

Supóngase que se quiere simular un ecosistema formado por tres especies (León, Ñu y Hierba), inicialmente en equilibrio. Este ecosistema sufre la invasión de un predador (Pantera) cuando el tiempo de la simulación vale 50. Este modelo se explica detalladamente en [Alfo98c] y en la sección X.3. La multimedia nos ha sido particularmente útil en este ejemplo, ya que se puede explicar mejor al alumno lo que está pasando en cada momento, que básicamente es que cuando el predador invade el ecosistema, se produce una disminución de la población de los predadores, ya que hay más competencia, así como una disminución de la población de herbívoros, ya que hay más predadores, y por consiguiente un aumento de la población de productores primarios, hasta que se llega a un estado de equilibrio oscilante [Alfo99f].

Queremos mostrar imágenes con la cadena alimenticia en cada momento, así como textos explicativos (Paso 1 del procedimiento de la figura VI.9).

Además, se quieren destacar tres intervalos en la simulación, al comienzo de los dos primeros, cambiaremos la imagen con la cadena alimenticia, además en los tres intervalos, se mostrará un texto explicativo diferente. Estos intervalos son:

- Desde el inicio hasta la invasión del predador (hay equilibrio estático). Que tiene como condición *OOCSMP*:

```
START ( (TIME >=0)&&(TIME<50) )
```

  Ya que es cuando *TIME* vale 50 cuando se produce la invasión del predador.

- Desde la invasión del predador hasta que se llega al equilibrio oscilante. Este momento se produce cuando *TIME > 50* y por ejemplo cuando la derivada de la población de predadores y herbívoros es negativa, que es cuando la población de todos ellos está disminuyendo. Es decir, la condición sería:

```
START ((TIME>=50)&&(Lion.XP<0)&&(Cheetah.XP<0)&&(Gnu.XP<0))
```

  Si bien hay que poner esta condición para el texto, para la imagen es posible poner ya *DEFAULT* ya que sólo identificamos dos momentos: antes y después de la invasión.

- Una vez que se ha llegado al equilibrio oscilante. Que se produce cuando no se produce ninguno de los dos casos anteriores, es decir que la condición *OOCSMP* sería la acción por defecto (*DEFAULT*).

Con esto hemos terminado el paso dos del procedimiento, sólo nos queda preparar las imágenes y los textos explicativos, y decidir en qué lugar de la interfaz de usuario los vamos a poner. En el presente ejemplo, hemos decidido ponerlos en el panel principal, en las casillas *E* y *NE*. El listado VI.7 muestra cómo quedaría el modelo *OOCSMP*. Las figuras VI.10 y VI.11 muestran el aspecto del *applet* en la primera y en la segunda condición.

```
TITLE Three species , invasion of predator
INCLUDE "Species.csm"
* Actual species
Species Cheetah("Cheet", "wcat.gif",  4,-.028,.0014,.0,50)
Species Lion   ("Lion",  "lion.gif",   2,-.02, .001)
```

```
Species Gnu     ("Gnu","bovin.gif",20,-.02,.0001,.016666666)
Species LGrass ("LGrass","leafs.gif",400, .01, 0, .0005)
Species ecosystem := Cheetah, Lion, Gnu, LGrass
ecosystem.STEP()
Cheetah.ACTION(Gnu,        1, 1)
Lion.ACTION    (Gnu,        1, 1)
Gnu.ACTION     (Lion,    - .6, 0)
Gnu.ACTION     (Cheetah,- .4, 1)
Gnu.ACTION     (LGrass,     1, 1)
LGrass.ACTION (Gnu,      - 1, 1)
*************************************************************************
*                              Timer and show data                      *
*************************************************************************
TIMER delta:=0.01,FINTIM:=900,PRdelta:=.5,PLdelta:=5
PLOT          [N], Cheetah.X, Lion.X, Gnu.X, LGrass.X, TIME
ICONICPLOT    [C], Cheetah.X, Lion.X, Gnu.X, LGrass.X
TEXTPANEL     [E],   START((TIME>=0)&&TIME<50), "inicio.txt",
                     START((TIME>=50)&&(Lion.XP<0)&&(Cheetah.XP<0)&&(Gnu.XP<0)),
                     "inv.txt",
                     DEFAULT,"equilib.txt"
IMAGEPANEL    [NE],  START((TIME>=0)&&TIME<50), "inicio.gif",
                     DEFAULT, "inv.gif",
METHOD ADAMS
```

Listado VI.7: Modelo de un ecosistema invadido por un predador, enriquecido con multimedia.

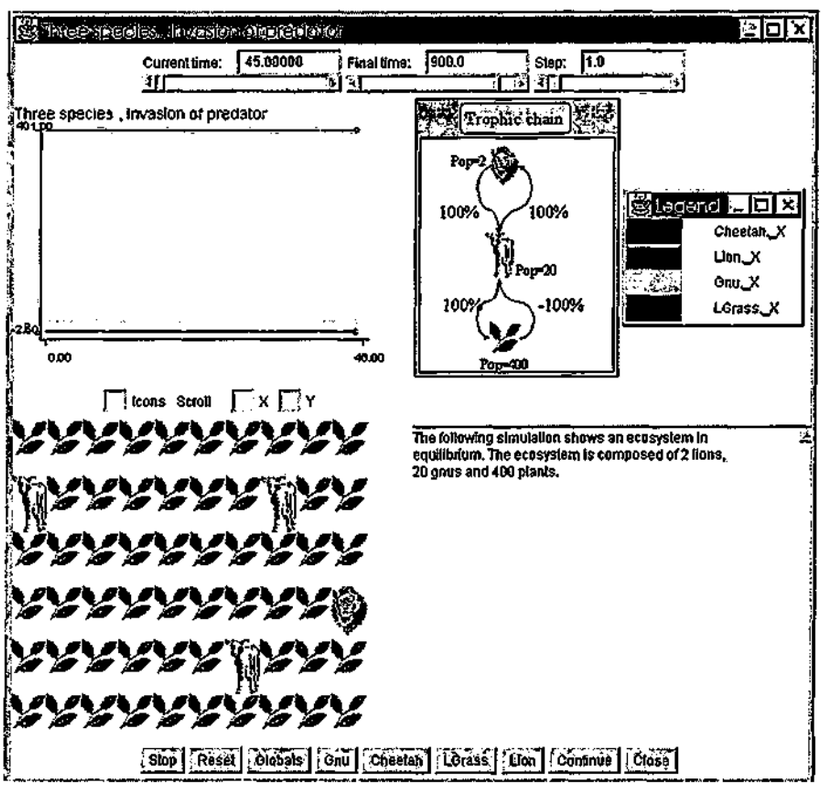


Figura VI.10: Simulación antes de la invasión del predador.

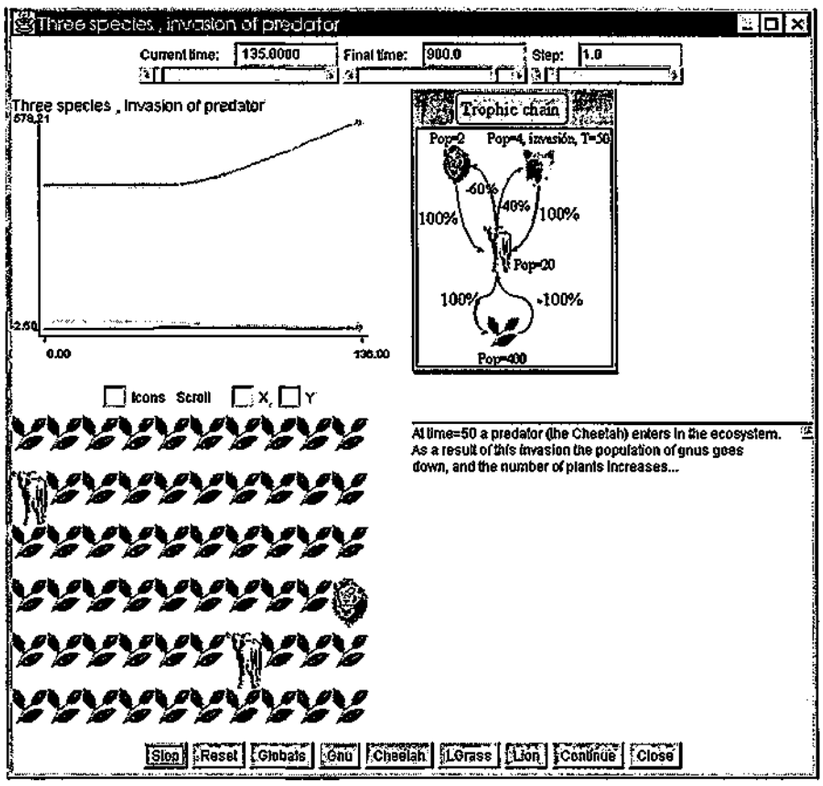


Figura VI.11: Simulación después de la invasión del predador.

■ VI.3.2 Simulación de un ecosistema con elementos multimedia (vídeo y audio)

En este problema vamos a simular varios ecosistemas simultáneamente, basándonos en las ecuaciones de Volterra [Volt31]. Cada ecosistema va a contener varias especies. Las especies pueden migrar de un ecosistema a otro. Se han implementado dos tipos de migraciones:

- Migración debido a un exceso de individuos de la misma especie. En este caso, el ecosistema destino se elige de entre todos los ecosistemas que existen. La especie que emigra elige el ecosistema con menos individuos de su misma especie, para minimizar la competencia; si bien se podía haber implementado algún otro criterio, tal como elegir el ecosistema con mayor número de individuos de las especies de las que se alimenta.

- Migración debido a la época del año. Para este tipo de emigración, el ecosistema destino es fijo. Se elige el primer ecosistema para el otoño, y el segundo para la primavera.

La información asociada a cada especie, tal como los coeficientes de preferencia alimenticia, el tipo de migración que realiza, etc., lo vamos a almacenar en una clase llamada *Species*. Otra clase, a la que hemos llamado *Population*, contiene el comportamiento de una determinada población de especies en un determinado ecosistema. Esta clase contiene información sobre el número inicial y actual de individuos, una referencia a la clase de especie a la que pertenece, etc.

Cada ecosistema se ha modelado como una colección de poblaciones de especies. Excepto para la migración, cada población solamente interacciona con las otras poblaciones de su mismo ecosistema.

Se han añadido dos elementos multimedia al modelo, ficheros de sonido que nombran la especie que emigra, y un vídeo explicando por qué se ha producido la migración (dos videos distintos, dependiendo del tipo de migración).

El momento en que se ha de poner el fichero de sonido es cuando se produce una migración:

- Para el caso de los Leones, que emigran debido al exceso de población, su sonido se dispara cuando: *(Li1.EMIGRAN # 0)||(Li2.EMIGRAN # 0)||(Li3.EMIGRAN # 0)*
- Para el caso de los Ñus, cuya emigración es debido a la estación, su sonido se dispara cuando:
  (Gn1.SPRING =0)||(Gn1.FALL=0)||(Gn2.FALL=0)||(Gn2.SPRING =0)||
  (Gn3.FALL=0)||(Gn3.SPRING =0).
- Para el caso de las Cebras, que también emigran debido al exceso de población, su sonido se dispara cuando:
  (Zb1.EMIGRAN # 0)||(Zb2.EMIGRAN # 0)||(Zb3.EMIGRAN # 0).

El vídeo se activa en los mismos momentos que el audio.

Para la simulación, se han creado tres ecosistemas. Uno de ellos está en equilibrio, y tiene tres especies (Leones, Ñus y Hierba). Los otros no lo están y tienen cuatro especies (Leones, Ñus, Cebras y Hierba). En la simulación se puede observar una invasión de Cebras al primer ecosistema, lo que rompe el equilibrio.

El listado VI.8 muestra el modelo *OOCSMP* del problema, y la figura VI.12 un momento de la simulación. En esta figura se puede observar como el ecosistema que estaba en equilibrio (el de arriba a la derecha, que se corresponde con el vector *Ecosystem1*) lo pierde en cierto momento, debido a una invasión de Cebras. Las Cebras vienen del ecosistema 2 (abajo a la izquierda). En el momento en que se produce la migración de este ecosistema, aumenta abruptamente la población de Hierba, que vuelve a decaer cuando aumenta la población de Ñus. También se puede observar el aumento de la población de Hierba en el tercer ecosistema (arriba a la izquierda en la figura), debido a la baja población de herbívoros.

```
* Type of migration constants
DATA EXCESS:=0, SEASON:=1

* ************************************************************************
* Definition of the Species class                                        *
* ************************************************************************

CLASS Species
{
* ***************************** Data ***********************************
 NAME name
 ICON spIcon
 DATA CANMIGRATE, TYPE
 DATA Percent[4], Last[4]
 DATA M, N1, N2:=0
}

CLASS Population
{
   NAME popName
   DATA X0, orden
   Species Sp

   DYNAMIC

    XT:=INTGRL(X0,XP)
    XP:=X*Sp.M
    X :=INSW(XT,0,XT)
    XPdel1 :=0
    XPdel2 :=0
    TEats  :=0
    TEats0 :=0

   DECX num
    X-=num
    XP-=num*Sp.M

   MIGRATE1 Population Ecos[]
    EXCESO := Sp.CANMIGRATE*(2*X0-X)
    EMIGRAN:= INSW(EXCESO,EXCESO,0)
    POSMIN := POSITION ( MIN ( Ecos.X ), Ecos.X )
    INSW (EMIGRAN,DECX(-EMIGRAN),)
```

```
    INSW (EMIGRAN,Ecos[POSMIN].DECX(EMIGRAN),)

   MIGRATE2 Population Ecosys[]
    SPRING := FCNSW(TIME*X, 1, 1, TIME%50)
    FALL := FCNSW(TIME*X, 1, 1, TIME%100)
    FCNSW (FALL,,INSW(Sp.CANMIGRATE,,Ecosys[0].DECX(-X)),)
    FCNSW (SPRING,,INSW(Sp.CANMIGRATE*FALL,,Ecosys[1].DECX(-X)),)
    FCNSW (FALL+SPRING,,DECX(X),)

   MIGRATE Population Others[], Population Places[]
    FCNSW ( Sp.TYPE, , MIGRATE1(Others), MIGRATE2(Places) )

* *************************** ACTION ********************************
ACTION Population S
 XPdel1 +=FCNSW(Sp.Percent[S.orden]*S.TEats, Sp.Percent[S.orden]*S.X*S.TEats0/S.TEats,
                                             0, 0)
 XPdel2 +=FCNSW(Sp.Percent[S.orden]*S.X0, 0, 0, Sp.Percent[S.orden]*S.X*S.X/S.X0)
 TEats  +=FCNSW(Sp.Percent[S.orden], 0, 0, 1)*S.X
 TEats0 +=FCNSW(Sp.Percent[S.orden], 0, 0, 1)*S.X0
 XP +=FCNSW(Sp.Percent[S.orden]*X0,  Sp.Last[S.orden]*Sp.N2*X*XPdel1*X/X0, 0,
                                     Sp.Last[S.orden]*Sp.N1*X*XPdel2*TEats0/TEats)
)

DATA PercLi[4], PercLi[] := 0 0.7 0.3 0
DATA LastLi[4], LastLi[] := 0   0   1 0

DATA PercGn[4], PercGn[] := -1 0 0 1
DATA LastGn[4], LastGn[] :=  1 0 0 1

DATA PercZb[4], PercZb[] := -1 0 0 1
DATA LastZb[4], LastZb[] :=  1 0 0 1

DATA PercLG[4], PercLG[] :=  0 -0.5 -0.5 0
DATA LastLG[4], LastLG[] :=  0     0   1 0

Species Lion   ("Lion","lion002.gif",1,EXCESS,PercLi,LastLi,-.0195, .0013982857142857 )
Species Gnu    ("Gnu","bovin008.gif",1,SEASON, PercGn, LastGn,-.02,.0002, .03 )
Species Zebra  ("Zebra","zebra003.gif",1,EXCESS,PercZb,LastZb,-.01,.0000625, .0075)
Species LGrass ("LGrass","leafs015.gif",0,EXCESS,PercLG,LastLG,.01,.0, 0.001)

Population Li1 ( "Lion1",  2,  0, Lion)
Population Gn1 ( "Gnu1",   20, 1, Gnu )
Population Zb1 ( "Zebra1", 0,  2, Zebra )
Population LG1 ( "LGrass1",400,3, LGrass )

Population Li2 ( "Lion2",  2,  0,  Lion)
Population Gn2 ( "Gnu2",   22, 1, Gnu )
Population Zb2 ( "Zebra2", 20, 2, Zebra )
Population LG2 ( "LGrass2",300,3, LGrass )

Population Li3 ( "Lion3",  5,  0,  Lion)
Population Gn3 ( "Gnu3",   3,  1, Gnu )
Population Zb3 ( "Zebra3", 34, 2, Zebra )
Population LG3 ( "LGrass3",720,3, LGrass )


Population Ecosystem1 := Li1, Gn1, Zb1, LG1
Population Ecosystem2 := Li2, Gn2, Zb2, LG2
Population Ecosystem3 := Li3, Gn3, Zb3, LG3

Population Lions := Li1, Li2, Li3
Population Gnus  := Gn1, Gn2, Gn3
Population Zebras:= Zb1, Zb2, Zb3

DYNAMIC
  NOSORT
  Lions.MIGRATE(Lions,Lions)
  Gnus.MIGRATE(Gnus,Gnus)
  Zebras.MIGRATE(Zebras,Zebras)
  Ecosystem1.STEP()
  Ecosystem1.ACTION(Ecosystem1)
  Ecosystem2.STEP()
  Ecosystem2.ACTION(Ecosystem2)
  Ecosystem3.STEP()
  Ecosystem3.ACTION(Ecosystem3)
```

```
PLOT  [C], Ecosystem3.X, TIME
PLOT  [E], Ecosystem1.X, TIME
PLOT  [S], Ecosystem2.X, TIME

AUDIOPANEL     START( (Li1.EMIGRAN#0)||(Li2.EMIGRAN#0)||
                      (Li3.EMIGRAN#0)),                          "Lion.wav",
               START( (Zb1.EMIGRAN#0)||(Zb2.EMIGRAN#0)||
                      (Zb3.EMIGRAN#0)),                          "Zebra.wav",
               START( (Gn1.SPRING =0)||(Gn1.FALL=0)||(Gn2.FALL=0)||(Gn2.SPRING =0)||
                      (Gn3.FALL=0)||(Gn3.SPRING =0)),            "Gnu.wav"
VIDEOPANEL     [SE],
               START( (Li1.EMIGRAN#0)||(Li2.EMIGRAN#0)||
                      (Li3.EMIGRAN#0)),                          "Lion.avi",
               START( (Zb1.EMIGRAN#0)||(Zb2.EMIGRAN#0)||
                      (Zb3.EMIGRAN#0)),                          "Zebra.avi",
               START( (Gn1.SPRING =0)||(Gn1.FALL=0)||(Gn2.FALL=0)||(Gn2.SPRING =0)||
                      (Gn3.FALL=0)||(Gn3.SPRING =0)),            "Gnu.avi"
TIMER delta:=0.01,FINTIM:=320,PRdelta:=.5,PLdelta:=5
```

Listado VI. 8: Modelo *OOCSMP* de varios ecosistemas con migraciones entre sí.

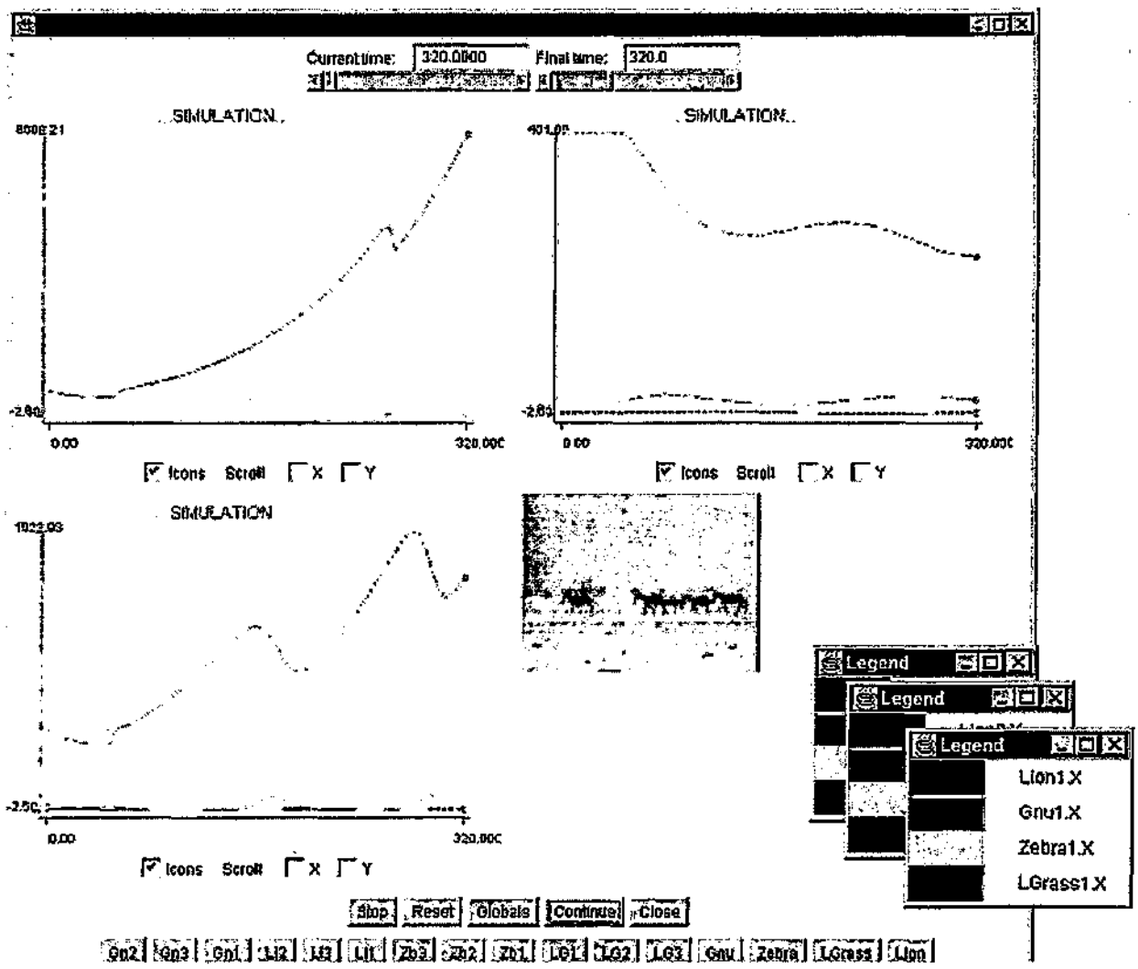


Figura VI.12: Simulación del modelo de ecosistemas con migraciones.

VII. Extensiones para la distribución

Esta sección presenta las extensiones que hemos añadido al lenguaje, y que hacen posible las simulaciones distribuidas [Alfo00a], [Alfo00b]. El capítulo se estructura como sigue:

La sección uno es la introducción, en la que se presenta una introducción a la simulación paralela y distribuida, un esquema de cómo se ha llevado a cabo la distribución en el lenguaje *OOCSMP,* así como una introducción a los mecanismos clásicos de sincronización, y a cómo se ha enfocado la sincronización en *OOCSMP.*

La sección dos presenta detalladamente cada una de las primitivas para la distribución, junto con ejemplos de su uso.

En la sección tres se muestran varios ejemplos de simulaciones distribuidas. El primero es un ejemplo que muestra cómo genera el compilador el código distribuido. Los siguientes son ejemplos reales: en la sección VII.3.2 se presenta el ejemplo del calentamiento de las vigas, visto en el apartado V.6.1; el siguiente es una variación más compleja del anterior, con piezas móviles que se juntan y se separan. El último es el de la simulación de ecosistemas con migraciones, presentado en la sección VI.3.2.

VII.1 Introducción

VII.1.1 Simulación paralela y distribuida. Simulación basada en la web.

Para sistemas complejos y grandes, la limitación a una ejecución secuencial puede ser severa. Resulta importante encontrar un formalismo adaptado a un entorno distribuido, en el que varias partes de la simulación se estén ejecutando concurrentemente. La simulación distribuida se conoce con las siglas *PADS* (Parallel And Distributed Simulation).

Internet está emergiendo como un entorno en el que muchas disciplinas están reevaluando sus técnicas y filosofías tradicionales [Page00]. Una de estas disciplinas es la simulación. Internet ofrece servicios útiles para la computación, tales como su arquitectura en red distribuida, y protocolos comunes para la comunicación de ordenadores. El número de computadores conectados en este entorno, hace posible sacar partido de su potencia de cómputo para resolver problemas más interesantes y complejos [JGra99].

Java es uno de los lenguajes más populares para Internet. Algunas de las propiedades que lo hacen interesante son el lema *"write once, run anywhere"*, la posibilidad de los programas Java de comunicarse entre sí por medio del formalismo Invocación de Método Remoto (Remote Method Invocation, *RMI*). *RMI* es el análogo a la llamada a procedimiento remoto (Remote Procedure Call, *RPC*) de la programación distribuida. En el modelo de objetos distribuidos de Java, un objeto remoto es aquel cuyos métodos se pueden invocar desde otra máquina virtual de Java, potencialmente en otro ordenador.

La computación basada en la Web (*Web computing*) es una clase especial de computación distribuida (cliente/servidor) en la que varios programas remotos colaboran través de Internet, usando protocolos estándar de Internet, y visores Web estándar, como bases para la interfaz de usuario [Serb97].

Tradicionalmente, en la simulación digital continua, los intentos de distribución se han centrado en paralelizar los diversos algoritmos de resolución de sistemas de ecuaciones, o bien las diversas operaciones con matrices [Kasc79], [JáJá92]. En nuestro contexto, este enfoque no es adecuado, debido a que normalmente implica un paralelismo de grano fino (por ejemplo en la resolución con métodos iterativos de problemas elípticos, cada procesador se asocia con un elemento de la matriz), y la red de interconexión de la que disponemos tiene una latencia muy alta, con lo que debemos tender a aumentar la granularidad, minimizando las comunicaciones y maximizando la computación.

La mayoría de los esfuerzos en cuanto a distribución, se han centrado en la simulación discreta. El conjunto de técnicas que tratan el paralelismo en simulación discreta se conoce como *PDES* (Parallel Discrete Event Simulation) [Zeig90]. Las técnicas *PDES* no se usan demasiado en aplicaciones de simulación comerciales, debido a su complejidad: el coste de la puesta a punto de un programa *PDES* es demasiado alto con respecto al beneficio que se obtiene [Page99].

Nuestro esquema de distribución es más natural y totalmente distinto de los anteriores. Aprovechamos la orientación a objetos para colocar los distintos objetos que toman parte en la simulación en diversas máquinas. De esta forma (idealmente) es posible ejecutar concurrentemente todo el comportamiento de los objetos. Este esquema es eficiente cuando se pueden localizar grupos de objetos que interaccionan mucho entre sí, y poco con otros grupos de objetos.

Es decir, nos encontraríamos en un enfoque *MIMD* (Multiple Instruction Multiple Data) según la clasificación de Flynn para las organizaciones paralelas, y además con memoria distribuida (modelo *NUMA*, Non Uniform Memory Access). En principio, cada procesador tiene acceso solamente a su espacio de direcciones, aunque como ya se ha comentado, la comunicación entre procesadores se realiza mediante *RMI*, siendo la latencia potencialmente muy elevada, ya que la red de interconexión es la red Internet. Si bien el problema de la latencia es grave, el uso de Internet como red de interconexión tiene una serie de ventajas que lo hacen interesante:

- Es una infraestructura que ya existe.

- Es altamente escalable.
- Protocolos y mecanismos de comunicación comunes y bien establecidos.

Este esquema de distribución permite acercarse al paradigma de '*interoperabilidad*'. El concepto de interoperabilidad en el contexto de la simulación, significa poder reutilizar componentes, usándolos en nuevos contextos, siendo posible añadir o eliminar componentes durante la ejecución de la simulación. Esta característica se ha añadido al estándar para aplicaciones distribuidas *HLA* (High Level Architecture) [Dham97], [HLA99] desarrollado por el Departamento de Defensa de los Estados Unidos. *HLA* se aplica fundamentalmente a simulaciones de tipo militar, en la que hay intervención y participación humana (*human-in-the-loop simulation*).

VII.1.2 Distribución en *OOCSMP*

Hemos extendido el lenguaje añadiendo instrucciones que hacen posible ejecutar simulaciones distribuidas. Estas extensiones, por el momento, sólo tienen efecto cuando se genera código Java, ya que la distribución se lleva a cabo mediante el paquete *rmi* [Berg97] de Java. También es necesario tener el protocolo *TCP/IP* instalado y funcionando en cada máquina participante en la simulación.

El uso del lenguaje Java y *rmi* nos va a posibilitar la integración de máquinas heterogéneas colaborando en la solución de una simulación, debido a que Java es multiplataforma.

Para realizar una simulación distribuida, no es necesario programar un modelo por cada máquina, el mismo modelo vale para todas las máquinas, pero se ha de compilar con *C-OOL* una vez por cada máquina que vaya a intervenir en la simulación, pasándole al compilador el nombre de la máquina como parámetro.

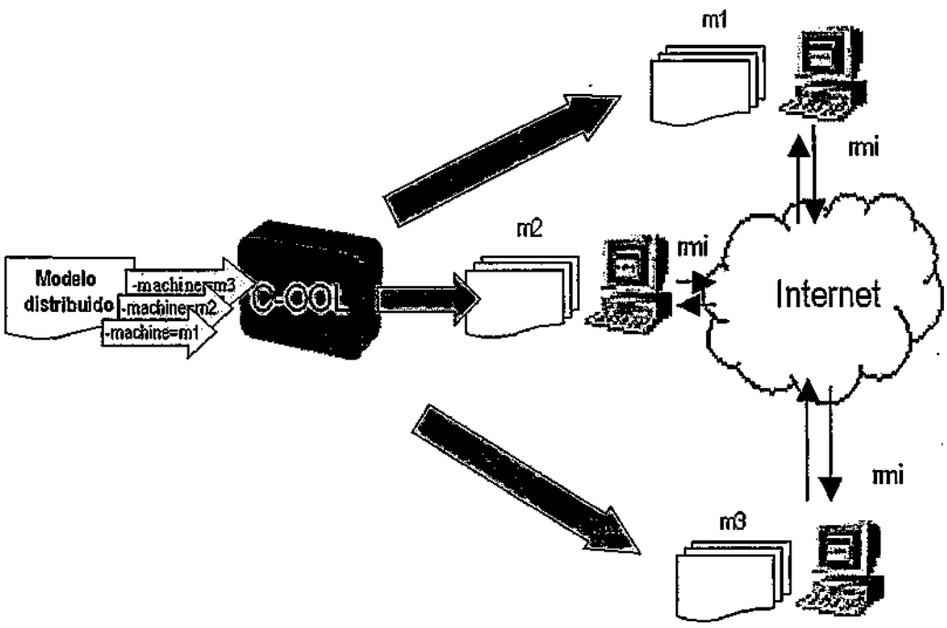


Figura VII. 1: Esquema de la distribución en *OOCSMP*

La distribución es a nivel de objeto, es decir, en la creación de un objeto se puede decidir en qué máquina se quiere crear, y por tanto en qué máquina se van a ejecutar las llamadas a los métodos de ese objeto. También existe la posibilidad de replicar un objeto en todas las máquinas, así como decidir en qué máquina va a mostrarse una determinada salida gráfica, y qué método de integración va a usarse en cada una.

Las variables globales se replican en todas las máquinas. Esta redundancia en el cómputo, en nuestro caso no es perjudicial, ya que estamos reemplazando comunicaciones (recuérdese el tiempo alto de latencia) por computación.

En cada máquina se ejecuta el bucle principal de simulación (sección *DYNAMIC* principal), de forma que cada máquina ejecuta los métodos de sus objetos locales. Las instrucciones que consisten en la ejecución de métodos de objetos remotos no los ejecuta, debido a que el compilador no ha generado esas instrucciones en el código de la simulación (las genera en el código de la máquina correspondiente). En su lugar es posible que el compilador genere un semáforo. Los métodos de objetos remotos que necesiten ser ejecutados porque, por ejemplo, van como parámetros de métodos de objetos locales, o aparecen en la parte derecha de una igualdad, se ejecutan mediante *rmi*. Esto es transparente para el programador. Es decir, no hay un control global del cómputo, el esquema de planificación de tareas es totalmente distribuido, de acuerdo con la naturaleza también distribuida de Internet.

Las máquinas que intervienen en la simulación se especifican mediante etiquetas antes de la compilación, utilizando primitivas del lenguaje (ver sección VII.2.1), y no pueden cambiarse luego en tiempo de ejecución de la simulación. Modificar el esquema de distribución es bastante sencillo, ya que basta con cambiar las máquinas asociadas a cada etiqueta y recompilar.

La distribución que se ha implementado es eficiente cuando existen pocas dependencias entre objetos que vayan a residir en distintas máquinas. Por ejemplo, en el problema de las cuatro vigas, del apartado V.6.1, la distribución es bastante eficiente, ya que se puede asignar cada objeto a una máquina distinta, y estos no tienen dependencias entre sí.

En la mayoría de los problemas, el coste de pasar de una simulación mono-máquina a una simulación distribuida es mínimo, ya que sólo se ha de indicar las máquinas que intervienen en la simulación y dónde se quiere crear cada objeto.

■ VII.1.3 Sincronización

En sistemas distribuidos existen recursos no compartibles, que no pueden ser usados simultáneamente por varios procesos concurrentes [Pete91]. Los procesos que compiten por este recurso han de sincronizarse (uno accede al recurso, y cuando termina lo libera, de forma que los restantes se ponen de acuerdo para determinar cuál es el siguiente que puede utilizarlo), ya que en otro caso se producirían errores. En el caso de la computación distribuida, estos errores se pondrían de manifiesto en que la solución del problema no sería la misma que en el caso de realizarse la computación de forma secuencial.

Una sección crítica es un conjunto de operaciones que sólo pueden ser ejecutadas a la vez por un proceso. La exclusión mutua es el acceso excluyente a una sección crítica por parte de un conjunto de procesos concurrentes. Los sistemas paralelos deben huir de situaciones de interbloqueo, que consisten en que un grupo de procesos está esperando permanentemente, por haber adquirido cada uno recursos que otros necesitan, y además necesitan recursos que posee alguno de los restantes y que no ha liberado.

Existen diversos enfoques con respecto al interbloqueo. El primero consiste en evitarlo, el segundo en deshacerlo, una vez que se produce. En este trabajo hemos tomado el enfoque de evitarlo.

Un mecanismo para regular el funcionamiento de un sistema paralelo consiste en establecer puntos de sincronización en cada proceso, para asegurar que sólo se ejecutarán las instrucciones que siguen si el resto de procesos ha llegado a su punto de sincronía. Otras alternativas son los monitores [Hans73] [Hoar74] (para restringir el acceso a ciertas variables) o el intercambio de mensajes (síncronos o asíncronos).

Para asegurar el acceso único a secciones críticas, que en nuestro caso son las llamadas a métodos de objetos, en el presente trabajo se ha optado por los semáforos [Dijk65]. Los semáforos son mecanismos de sincronización que se asocian a un determinado recurso. Cuando un proceso va a usar el recurso, comprueba el estado de su semáforo asociado. Si este tiene cierto valor que permite su uso, el proceso coloca el semáforo en un estado que bloquee al resto de procesos, y se apropia del recurso. Cuando termina su uso, coloca el semáforo en un estado que habilite a uno de los procesos que esperan.

En *OOCSMP*, los semáforos los pone automáticamente el compilador. Por defecto, también se pone un punto de sincronización entre todas las máquinas, al principio y al final de cada paso elemental de tiempo, aunque tanto los semáforos como esta sincronización se pueden desactivar mediante opciones de

compilación. Los semáforos garantizan la serialización de la simulación distribuida, es decir, que el resultado sea el mismo que el de una simulación en una sola máquina.

Hay dos clases de puntos de sincronización:

- Puntos de sincronización para mantener el mismo tiempo de simulación en todas las máquinas.
- Semáforos para la sincronización entre objetos.

A continuación se detalla brevemente la implementación de cada clase de semáforo. Esto es totalmente transparente al constructor del modelo *OOCSMP*. Los semáforos se han implementado como métodos Java que genera el compilador de forma automática.

VII.1.3.1 Puntos de sincronización del tiempo de simulación

Para la implementación de este tipo de sincronización, en cada máquina se tiene una variable con el tiempo actual de simulación en el resto de las máquinas. El punto de sincronización es generado automáticamente por el compilador en una función Java llamada `wait_until_TIME()`, cuyo pseudocódigo es el siguiente:

```
wait_until_TIME()
{
      para el resto de las máquinas de la simulación
        llamar a la función remota finishLoop<nombre-maquina-local>(TIME)
      fin_para
      mientras ( allEqualsTIME() == falso ) hacer
      fin_mientras
}
```

Listado VII.1: Esquema de los semáforos de sincronización de tiempo

Además se crea una función llamada `finishLoop<maquina>` para cada máquina distribuida. Esta función guarda en la variable `TIME<maquina>` el valor del argumento de la función. La función `allEqualsTIME` devuelve `TRUE` si el tiempo local y el de cada una de las máquinas distribuidas difieren en menos que $\Delta/2$, o bien en menos que $3\Delta/2$ y además el tiempo local es el menor. Es decir, la máquina más rápida ha de esperar al final de cada bucle de simulación sin hacer cómputo.

VII.1.3.2 Semáforos para la sincronización de objetos

Si se genera código distribuido, en cada clase se crea automáticamente una variable entera llamada `semaphore`, que se incrementa al final de cada método del objeto, y se pone a cero al inicio de cada pasada del bucle de simulación. Esta variable es un indicador del lugar del bucle de simulación del modelo secuencial en el que se ejecuta cada método.

Para cada objeto que resida en la máquina, el compilador crea una función llamada `get<nombre-objeto>`, esta función recibe dos parámetros:

- Un parámetro (t) de tipo `double`, que representa el tiempo en la máquina remota.
- Un parámetro (s) de tipo entero, que representa el número de veces que se tiene que haber llamado a un método de ese objeto para que la función pueda devolver dicho objeto.

Esta función espera a que el tiempo local y el parámetro t difieran en menos de Δ, y entonces espera a que el valor de la variable `semaphore` del objeto sea igual que s. Esto no produce una situación de interbloqueo, porque el semáforo se ejecuta en un *thread* distinto al bucle de simulación (el proceso que solicita el objeto puede tener que esperar, el proceso poseedor del objeto no espera, debido a que el código del semáforo se ejecuta en un *thread* aparte). Además se crean semáforos similares para todos los métodos y atributos que vayan a ser accedidos desde otra máquina. Estas funciones se denominan `get<metodo>_<objeto>` y en lugar de devolver el objeto, ejecutan el método `<metodo>` sobre el objeto `<objeto>` y devuelven el resultado. Los atributos accedidos desde otra máquina se devuelven a través de un semáforo llamado `get<atributo>_<objeto>`, este semáforo además incrementa la

variable `semaphore`, como si se hubiera llamado a un método del objeto. En el primer ejemplo se ve claramente la razón de esto.

Si se accede a un método, atributo u objeto entero que es remoto, se llama a este semáforo mediante *rmi* en la máquina remota en la que reside dicho objeto. El control se nos devuelve en el instante en que la simulación en la máquina remota alcance el tiempo en que lo ha pedido la máquina local, y además, en el momento adecuado del bucle de simulación (que viene dado por el segundo parámetro). Esto garantiza la serialización.

Si el objeto es local, se accede al semáforo en la máquina local, esto es necesario, porque puede ser que el objeto se esté usando en una máquina remota.

Como se tiene un único programa *OOCSMP*, al analizar el bucle principal, el compilador conoce en qué momento se ha de pedir cada objeto remoto (sabe qué argumento poner en la llamada a `get<nombre-objeto>`), pues tiene información de qué métodos se deben haber ejecutado en la simulación secuencial, y además también conoce en qué máquina reside el objeto.

Como ya se ha indicado con anterioridad, todos estos mecanismos de sincronización son totalmente transparentes para el usuario. Un modelo distribuido *OOCSMP* es igual que un modelo secuencial, salvo quizá en la parte declarativa, en la que se elige la máquina en que se va a crear cada objeto, y en la parte de declaración de variables de control, en la que se elige la salida gráfica para cada máquina.

VII.2 Extensiones para la distribución

En esta sección se van a presentar todas las extensiones que se han añadido a *OOCSMP* para hacer posible la distribución.

VII.2.1 Etiquetado de máquinas

En un modelo distribuido, es necesario declarar todas las máquinas que van a intervenir en la simulación. En esta declaración se las etiqueta con un nombre, que será usado en todas las referencias posteriores a esa máquina. Esta primitiva recibe el nombre de *MACHINE*, y tiene la siguiente sintaxis:

```
MACHINE <identificador> "<nombre maquina>"
```

Sintaxis VII.1: Sintaxis del etiquetado de máquinas

Donde *<identificador>* es la etiqueta con que se referenciará la máquina *<nombre maquina>* en todas las primitivas de distribución. *<nombre maquina>* es la dirección *ip* o bien el nombre Internet de la máquina. Es posible etiquetar todas las máquinas en un fichero aparte y luego incluirlo, mediante la instrucción *INCLUDE*. De esta forma es fácil variar el esquema de distribución. Una misma máquina puede tener etiquetas distintas.

VII.2.2 Creación de objetos

En un modelo distribuido, es necesario indicar en qué máquina se va a crear el objeto, o bien si éste va a ser replicado en todas las máquinas que intervengan en la simulación. La declaración de un objeto indica dónde se va a crear. La sintaxis es la siguiente:

```
<Clase> <objeto> ( <parametros> ) MACHINE <id_maquina>
```

Sintaxis VII.2: Sintaxis para la distribución de objetos en máquinas

De esta forma, el objeto *<objeto>* sólo existe en el modelo que residirá en la máquina *<id_maquina>*. Las llamadas a métodos de ese objeto se ejecutarán en esa máquina. Si un objeto va a ser replicado en todas las máquinas, basta con omitir *MACHINE <id_maquina>*.

VII.2.3 Colecciones de objetos

Las colecciones de objetos se replican en todas las máquinas, pero sólo contienen los objetos que residan en la máquina local. El compilador se ocupa de que las iteraciones sobre estas colecciones se realicen en el orden adecuado. Este vector se utiliza cuando es el receptor de cualquier método. Insertando sólo los objetos locales en el vector, nos aseguramos de que los métodos se ejecuten sólo sobre objetos locales.

Cuando el vector no es el receptor de un método, sino que se utiliza como parámetro de algún método, en lugar de usar el vector, se llama a una función que busca cada objeto del vector, ya sea éste local o remoto.

VII.2.4 Método de integración

Es posible variar el método de integración que se va a usar en cada máquina, debido a que la instrucción *METHOD* tiene ahora la siguiente sintaxis:

```
METHOD [ [MACHINE=<id>],] <metodo_intgr>
       ( ,[MACHINE=<id>], <metodo_intgr> )*
```

Sintaxis VII.3 : Sintaxis de la instrucción *METHOD*.

Si no se especifica ningún nombre de máquina, el método de integración especificado se aplica a todas las máquinas de la simulación.

VII.2.5 Salidas gráficas

Es posible indicar en qué máquina se quiere que aparezca cada una de las salidas gráficas. También se puede monitorizar objetos remotos, es decir, hacer que una salida gráfica en una máquina presente información sobre datos que están en otra máquina, aunque esto es poco eficiente.

La máquina en la que se quiere la salida gráfica se indica en el parámetro *[MACHINE=<maquina>]* de las salidas gráficas (ver sección VI).

VII.3 Ejemplos de simulaciones distribuidas

VII.3.1 Generación de código distribuido

En esta sección se va a presentar un ejemplo de cómo generaría el compilador código distribuido. Supongamos que tenemos el modelo del listado VII.2, en el que se declaran dos objetos *A* y *B*, ambos de clase *C*. El objeto *A* se crea en la máquina *m1*, y el objeto *B* en *m2*.

```
MACHINE m1 <dirección-ip>
MACHINE m2 <dirección-ip>

CLASS C
{
  ATRIBUTO1
  ...
  METODO1 ...
  ...
  METODO2  PARAM
  ...
}

C A(...) MACHINE m1
C B(...) MACHINE m2
DYNAMIC
  A.METODO1(...)
  B.METODO1(...)
  A.METODO2 (B.ATRIBUTO1)
  B.METODO2 (...)
TIMER ...
```

Listado VII.2: Esquema del ejemplo propuesto.

La clase *C* tiene dos métodos (*METODO1* y *METODO2*) y un atributo (*ATRIBUTO1*). El punto crítico es la instrucción que invoca *METODO2* sobre el objeto *A*, pasando como parámetro un atributo del objeto *B*, remoto. El compilador debe asegurarnos que cuando vayamos a acceder a dicho atributo, el objeto *B* esté en el mismo estado que en el caso de la simulación secuencial. El código generado para cada máquina tendría el siguiente aspecto:

MÁQUINA M1

```
CREAR EL OBJETO A
BUCLE PRINCIPAL DE SIMULACION
SINCRONIZAR TIEMPO
(1) GETA(TIME,0).METODO1(...)
(2) GETA(TIME,1).METODO2(
   M2.getATRIBUTO1_B(TIME,1))
SINCRONIZAR TIEMPO
```

MÁQUINA M2

```
CREAR EL OBJETO B
BUCLE PRINCIPAL DE SIMULACIÓN
SINCRONIZAR TIEMPO
(1) GETB(TIME,0).METODO1(...)
(2) GETB(TIME,2).METODO2(...)
SINCRONIZAR TIEMPO
```

Listado VII.3: Esquema del código generado.

Donde la notación *GETA*, o *GETB* indica que estamos llamando al semáforo de *A* o *B* en la máquina local. *M2.getATRIBUTO1_B* significa la llamada mediante *rmi* al semáforo de la máquina *M2* que devuelve el atributo *ATRIBUTO1* del objeto *B*.

Ambas máquinas empiezan la ejecución llamando al semáforo local sobre *A* y *B*. Se requiere que tanto *A* como *B* hayan ejecutado cero métodos (parámetro 0). Ambos semáforos retornan inmediatamente.

La instrucción (2) de la máquina *M1*, llama al semáforo de *A*, que retorna inmediatamente, y además llama mediante *rmi* al semáforo de *B*. El objeto *B* es devuelto después de haber ejecutado (1) la máquina *M2*, ya que *M1* pide *B* con el semáforo con valor 1. El atributo *ATRIBUTO1* se pide mediante un método, que hace incrementar la variable *semaphore* de *B* en 1. Esta es la razón de que la instrucción (2) de la máquina *M2* pida el objeto *B* con *semaphore=2*: es necesario que *M1* haya usado al atributo *atributo1* antes de que *M2* pueda ejecutar el método *METODO2* sobre *B*, ya que este método podría cambiar el valor del atributo *atributo1*.

Es decir, las instrucciones (1) de las dos máquinas se pueden ejecutar a la vez, pero la instrucción (2) de *M1* se ha de ejecutar antes que la (2) de *M2*.

■ VII.3.2 Cálculo del calor en cuatro vigas

En esta sección vamos a ampliar el modelo de la sección V.6.1, para hacerlo distribuido. En este caso, el esquema de distribución es obvio: colocar cada objeto en una máquina distinta. Teniendo en cuenta que no hay interacciones entre los objetos, esta distribución es óptima, siendo el coste del cambio del modelo mínimo, como se puede ver en los listados VII.4 y VII.5.

```
TITLE Distributed simulation of the heating of 4 pieces

INCLUDE "machines.csm"

CLASS Piece
{
    * Define class parameters, translation of the piece components
    DATA trx1, try1, trx2, try2
    * Conductivity of the material
    DATA Kx, Ky

    DOMAIN qd1 := QUADRILATERAL(0, 0, 2, 0, 2, 1, 0, 1
                  , INITIAL(0)
      ,ESSENTIAL(EDGE(1:4),EXP(2.0*TIME)*SIN(X+Y)*CH(X+Y))
     )
    DOMAIN qd2 := QUADRILATERAL(0,-1, 1,-1, 1, 0, 0, 0
    , INITIAL(0)
     , ESSENTIAL(EDGE(1:4),EXP(2.0*TIME)*SIN(X+Y)*CH(X+Y))
     )

    qd1.TRANSLATE (trx1, try1)
    qd2.TRANSLATE (trx2, try2)

    * Mesh the domains
    MESH m1 := ISOPARAMETRIC (qd1, QUADRILAT4, ELEMENTS(100,50) )
```

```
    MESH m2 := ISOPARAMETRIC (qd2, QUADRILAT4, ELEMENTS(50 ,50) )

    * Define the Heat equation in 2d
    PDE H2da(0, 0, 1, 1,-Kx, 1, -Ky, 0, 0, 0, 0, 0, 0, 0, 0, FEM )
    PDE H2db(0, 0, 1, 1,-Kx, 1, -Ky, 0, 0, 0, 0, 0, 0, 0, 0, FEM )
    m1.CONCAT(m2)
    m1.setPDE(H2da)
    m2.setPDE(H2db)

    DYNAMIC
    m1.STEP()
}

Piece p1 (0, 0, 0, 0, 3.0, 3.0 )     MACHINE m1
Piece p2 (3, 0, 4, 0, 3.5, 4.0 )     MACHINE m2
Piece p3 (0, -4, 0, -2, 3.0, 3.4 ) MACHINE m3
Piece p4 (3, -4, 4, -2, 3.1, 3.1 ) MACHINE m4


DYNAMIC
Piece.STEP()

TIMER FINTIM:=1.0, delta:=0.05,PLdelta:=0.1
ISOPLOT   [C], [MACHINE=m1], p1.m1
ISOPLOT   [C], [MACHINE=m2], p2.m1
ISOPLOT   [C], [MACHINE=m3], p3.m1
ISOPLOT   [C], [MACHINE=m4], p4.m1
```

Listado VII.4: Modelo distribuido del problema V.6.1.

```
* Declaration of the machine names
MACHINE m1 "urano.ii.uam.es"
MACHINE m2 "minerva.ii.uam.es"
MACHINE m3 "kronos.ii.uam.es"
MACHINE m4 "afrodita.ii.uam.es"
```

Listado VII.5: Máquinas usadas para la simulación (fichero *machines.csm*).

En el presente ejemplo, hemos utilizado tres estaciones UNIX (*m2*, *m3* y *m4*) y una máquina con Windows'95 (*m1*). Hemos creado un objeto en cada máquina, así como una salida gráfica con el resultado de la simulación del propio objeto.

Se ha compilado el programa con la opción *-NOSEMAPHORES* activada en todas las máquinas, para no generar ningún tipo de sincronización entre las máquinas, debido a que no hace falta ninguna sincronización en los pasos de tiempo. En el presente ejemplo, el tiempo que tarda la simulación lo marca la máquina más lenta, en este caso, *m3*.

Portar el presente ejemplo a otro esquema de distribución es sencillo: basta con cambiar las máquinas del fichero *machines.csm* y recompilar.

■ VII.3.3 Cálculo del calor en dos vigas móviles

Este problema es una variación del anterior, pero durante el bucle de simulación las piezas se mueven hasta llegar a juntarse, permanecen de este modo durante un rato, y luego se separan. Las piezas vuelven a acercarse cuando su distancia es mayor que un cierto valor. Además el problema se ha restringido a las dos piezas superiores.

Este problema es más complicado que el anterior, debido a que, cuando las piezas están separadas, se pueden calcular por separado, pero cuando se juntan, las mallas han de concatenarse, y la ecuación ha de calcularse en la malla resultante. Cuando se separan, vuelve a haber dos mallas, que pueden volver a calcularse por separado.

El esquema de distribución que adoptaremos es crear una pieza en cada máquina. A veces las dos piezas se van a poder calcular por separado, mientras que otras veces (cuando estén concatenadas), se van a tener que calcular en un solo procesador.

Para ello, vamos a subclasificar la clase anterior, para que el método *STEP* de la clase acepte dos parámetros, que serán las translaciones *X* e *Y* que se van a aplicar a la pieza. En nuestro problema, ambas piezas van a trasladarse de forma paralela al eje *X*. Vamos a suponer que la clase anterior la hemos guardado en un fichero llamado *Piece.csm.*

```
INCLUDE "machines.csm"
INCLUDE "Piece.csm"
CLASS MLPiece : LPiece
{
    DYNAMIC x, y
      m1.MOVE(x,y)
      m1.STEP()
}
DATA EPS := 1e-4, ACER := 1, JOINED := 0, MAXJOINTIME := 6

MLPiece p1 (0, 0, 0, 0, 3.0, 3.0 ) MACHINE m1
MLPiece p2 (3, 0, 4, 0, 3.5, 4.0 ) MACHINE m2

doJOIN
  p1.m1.CONCAT(p2.m1)
  ACER *= -1
  JOINED := MAXJOINTIME

doMOVE
  JOINED -= 1
  p1.STEP( ACER*0.05, 0 )
  p2.STEP( -ACER*0.05,0 )

doDETACH
  p1.m1.DETACH(p2.m1)
  doMOVE

DYNAMIC
  DIST := ABS(p1.m1.RIGHTX-p2.m2.LEFTX)
  JOINED-=1
  INSW ( (DIST+EPS)*(EPS-DIST), FCNSW(JOINED, doJOIN, , ), doMOVE)
  FCNSW( JOINED, , doDETACH, p1.m1.STEP())
  INSW ( DIST-2, , ACER*=-1)

TIMER FINTIM := 20, delta := 0.1, PLdelta := 0.1
ISOPLOT   [C], [MACHINE=m1], p1.m1
ISOPLOT   [C], [MACHINE=m2], p2.m1
```

Listado VII.6: Calentamiento de piezas móviles.

Básicamente, el programa comprueba la distancia entre las dos piezas, y si es menor que un umbral, concatena las mallas. Como se vio en la sección sobre *PDEs* (sección V.3.4), esta concatenación es más costosa que la concatenación de mallas que se produce en la sección de inicialización, ya que la malla resultante tiene que reconfigurar sus matrices, copiar la solución encontrada por la segunda malla, el valor de los jacobianos de los elementos de la segunda malla, etc.

El tiempo que las mallas van a estar concatenadas lo controla la variable *JOINED*, que se decrementa en uno en cada pasada del bucle de simulación, y cuando se hace cero, las mallas se separan. Cuando la distancia entre mallas es mayor que dos, la dirección de su movimiento se invierte.

Los puntos críticos en la sincronización son:

- El momento en que se concatenan ambas mallas, porque la información sobre la solución encontrada en la segunda malla tiene que viajar de un ordenador a otro, y desde ese momento sólo el primer ordenador tiene que realizar cálculos, mientras que el otro espera.

- La separación de ambas mallas, momento en que la solución encontrada en la segunda malla ha de volver desde el primer ordenador hasta el segundo. Desde este momento, hasta la próxima concatenación, ambos ordenadores pueden resolver las ecuaciones separadamente.

También se consigue una eficiencia alta con este problema, debido a que, durante la mayor parte del tiempo, ambas mallas pueden calcularse simultáneamente.

Un momento en la simulación del problema anterior en una de las máquinas (*m1*) se muestra en la figura VII.2. La figura se ha tomado en el momento en que ambas mallas están concatenadas, de forma que en las gráficas aparece la solución de las dos piezas, porque se están resolviendo como una sola.

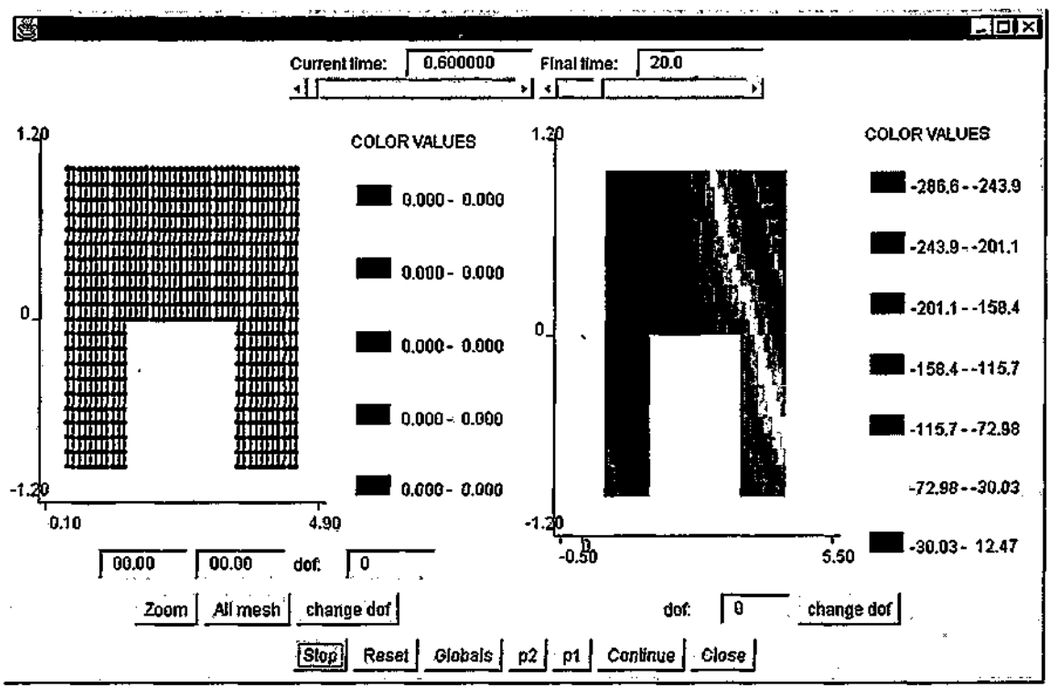


Figura VII. 2: Simulación del calor en dos piezas móviles.

Como se puede observar en la figura, se generan botones para cambiar ambos objetos, *p1* y *p2*, si bien, como ya se dijo, el cambio del objeto remoto (*p2*) es poco eficiente.

■ VII.3.4 Ecosistemas distribuidos con migraciones

En este ejemplo se ha modificado el modelo del apartado VI.3.2 para hacerlo distribuido. Vamos a poner cada ecosistema en una máquina distinta. Todas las poblaciones del mismo ecosistema estarán en la misma máquina. Además, vamos a replicar las clases con la información sobre las especies en todas las máquinas, ya que los objetos de la clase *Population* acceden constantemente a esta información, que además, es estática.

Las salidas gráficas de cada ecosistema las vamos a poner en su máquina correspondiente. También hay que modificar las señales de audio y vídeo, para que en cada máquina se comprueben sólo las migraciones del ecosistema local.

El modelo distribuido en una primera aproximación, queda muy parecido al modelo mono-máquina del listado VI.8:

```
INCLUDE "machines.csm"
DATA EXCESS:=0, SEASON:=1

... clases Specie y Popuation como en el listado VI.8
... igual delcaración de los vectores de porcentajes

Species Lion   ("Lion","lion002.gif",1,EXCESS,PercLi,LastLi,-.0195, .0013982857142857 )
Species Gnu    ("Gnu","bovin008.gif",1,SEASON, PercGn, LastGn,-.02,.0002, .03 )
Species Zebra  ("Zebra","zebra003.gif",1,EXCESS,PercZb,LastZb,-.01,.0000625, .0075)
Species LGrass ("LGrass","leafs015.gif",0,EXCESS,PercLG,LastLG,.01,.0, 0.001)

Population Li1 ( "Lion1",  2,  0, Lion)      MACHINE m1
Population Gn1 ( "Gnu1",   20, 1, Gnu )      MACHINE m1
Population Zb1 ( "Zebra1", 0,  2, Zebra )    MACHINE m1
Population LG1 ( "LGrass1",400,3, LGrass )   MACHINE m1

Population Li2 ( "Lion2",  2,  0,  Lion)     MACHINE m2
Population Gn2 ( "Gnu2",   22, 1, Gnu )      MACHINE m2
Population Zb2 ( "Zebra2", 20, 2, Zebra )    MACHINE m2
Population LG2 ( "LGrass2",300,3, LGrass )   MACHINE m2

Population Li3 ( "Lion3",  5,  0,  Lion)     MACHINE m3
Population Gn3 ( "Gnu3",   3,  1, Gnu )      MACHINE m3
Population Zb3 ( "Zebra3", 34, 2, Zebra )    MACHINE m3
Population LG3 ( "LGrass3",720,3, LGrass )   MACHINE m3

Population Ecosystem1 := Li1, Gn1, Zb1, LG1
Population Ecosystem2 := Li2, Gn2, Zb2, LG2
Population Ecosystem3 := Li3, Gn3, Zb3, LG3

Population Lions := Li1, Li2, Li3
Population Gnus  := Gn1, Gn2, Gn3
Population Zebras:= Zb1, Zb2, Zb3

DYNAMIC
  NOSORT
  Lions.MIGRATE(Lions,Lions)
  Gnus.MIGRATE(Gnus,Gnus)
  Zebras.MIGRATE(Zebras,Zebras)
  Ecosystem1.STEP()
  Ecosystem1.ACTION(Ecosystem1)
  Ecosystem2.STEP()
  Ecosystem2.ACTION(Ecosystem2)
  Ecosystem3.STEP()
  Ecosystem3.ACTION(Ecosystem3)

PLOT  [C], [MACHINE=m1], Ecosystem1.X, TIME
PLOT  [C], [MACHINE=m2], Ecosystem2.X, TIME
PLOT  [C], [MACHINE=m3], Ecosystem3.X, TIME
AUDIOPANEL     [MACHINE=m1],  START (Li1.EMIGRAN#0), "Lion.wav",
                              START (Zb1.EMIGRAN#0), "Zebra.wav",
                              START (Gn1.FALL=0||Gn1.SPRING=0),  "Gnu.wav"
AUDIOPANEL     [MACHINE=m3],  START (Li2.EMIGRAN#0), "Lion.wav",
                              START (Zb2.EMIGRAN#0), "Zebra.wav",
                              START (Gn2.FALL=0||Gn2.SPRING=0),  "Gnu.wav"
AUDIOPANEL     [MACHINE=m3],  START (Li3.EMIGRAN#0), "Lion.wav",
                              START (Zb3.EMIGRAN#0), "Zebra.wav",
                              START (Gn3.FALL=0||Gn3.SPRING=0),  "Gnu.wav"
VIDEOPANEL     [WINDOW], [MACHINE=m1],
                              START (Li1.EMIGRAN#0), "Lion.wav",
                              START (Zb1.EMIGRAN#0), "Zebra.wav",
                              START (Gn1.FALL=0||Gn1.SPRING=0),  "Gnu.wav"
VIDEOPANEL     [WINDOW], [MACHINE=m2],
                              START (Li2.EMIGRAN#0), "Lion.wav",
                              START (Zb2.EMIGRAN#0), "Zebra.wav",
                              START (Gn2.FALL=0||Gn2.SPRING=0),  "Gnu.wav"
VIDEOPANEL     [WINDOW], [MACHINE=m3],
                              START (Li3.EMIGRAN#0), "Lion.wav",
                              START (Zb3.EMIGRAN#0), "Zebra.wav",
                              START (Gn3.FALL=0||Gn3.SPRING=0),  "Gnu.wav"

TIMER delta:=0.005,FINTIM:=325,PRdelta:=.5,PLdelta:=2.5
METHOD RECT
```

Listado VII.7: Modelo distribuido del problema VI.3.2.

El fichero *machines.csm* es el del listado VII.5. Esta aproximación no es todo lo eficiente que podía ser, ya que, en cada pasada del bucle de simulación, llamamos al método *MIGRATE* con dos colecciones de objetos que contienen todos los objetos de un tipo de población, incluso los que son remotos. Esto da lugar a una serie de bloqueos innecesarios, que ralentizan la simulación. Una mejora consistiría en comprobar en cada caso si se emigra o no, y si es así, entonces llamar al método pasando como parámetros los vectores con todas las poblaciones. La solución sería añadir los siguientes métodos a la clase `Population` y cambiar el bucle principal de simulación:

```
CLASS Population
{
...
hasToMIGRATE1
   EXCESO1 := Sp.CANMIGRATE*(2*X0-X)
   hasToMigrate1:= INSW(EXCESO1,EXCESO1,0)

hasToMIGRATE2
     hasToMIGRATE2 := FCNSW(TIME%50, 0, 1, 0)

hasToMIGRATE
     hasToMIGRATE:=FCNSW ( Sp.TYPE, 0, hasToMIGRATE1, hasToMIGRATE2 )
}
DYNAMIC
     INSW(Li1.hasToMIGRATE, , Li1.MIGRATE(Lions, Lions) )
     INSW(Li2.hasToMIGRATE, , Li2.MIGRATE(Lions, Lions) )
     INSW(Li3.hasToMIGRATE, , Li3.MIGRATE(Lions, Lions) )
     INSW(Gn1.hasToMIGRATE, , Li1.MIGRATE(Gnus, Gnus) )
     INSW(Gn2.hasToMIGRATE, , Li2.MIGRATE(Gnus, Gnus) )
     INSW(Gn3.hasToMIGRATE, , Li3.MIGRATE(Gnus, Gnus) )
     INSW(Zb1.hasToMIGRATE, , Li1.MIGRATE(Zebras, Zebras) )
     INSW(Zb2.hasToMIGRATE, , Li2.MIGRATE(Zebras, Zebras) )
     INSW(Zb3.hasToMIGRATE, , Li3.MIGRATE(Zebras, Zebras) )
     ...
```

Listado VII.8: Modificación al modelo distribuido anterior.

La particularidad de ambas implementaciones está en las colecciones *Lions*, *Gnus* y *Zebras*. Como ya se mencionó, cuando estos vectores aparecen en la parte derecha de unas asignación, en realidad al generar el código distribuido, no se implementan como vectores, sino como funciones, que devuelven un objeto local, o llaman a un objeto remoto, según sea el caso. Sin embargo, cuando aparecen en la parte izquierda, o como receptores de métodos, sí que son vectores, que contienen únicamente los objetos locales.

VIII. Generación de documentos *HTML*

En este capítulo se presentarán las técnicas empleadas para la generación de documentos *HTML*. Estos documentos pueden ser bien documentación del modelo, o bien las páginas del curso que se está construyendo.

Comienza este capítulo con una introducción a la metodología que se ha seguido para facilitar la creación de cursos basados en simulación. En la sección VIII.1.1 se realiza un pequeño análisis de las herramientas de autor existentes en la actualidad. La sección 2 presenta las extensiones *OOCSMP* para la generación de páginas *HTML*. Estas extensiones se conocen como *SODA*. La sección 3 presenta las instrucciones para la generación automática de documentación del modelo

■ VIII.1 Introducción

Hasta ahora, en el presente trabajo se ha presentado un nuevo lenguaje para la descripción de modelos de simulación. Para el cumplimiento de nuestros objetivos, que son facilitar la construcción de cursos educativos para Internet, se hace necesario un medio para obtener fácilmente las páginas *HTML* en las que se va a incluir dicho modelo de simulación.

Para lograr este objetivo, se ha añadido al lenguaje *OOCSMP* instrucciones que permiten definir la apariencia de las páginas del curso. Se ha llamado *SODA* (Simulation COurse Description LAnguage) a este subconjunto de la gramática de *OOCSMP*, porque normalmente no se mezclan en un mismo fichero las instrucciones para la descripción del modelo, con las instrucciones para la descripción de la página en la que se incluirá. Se podría considerar a *SODA* como un lenguaje de nivel distinto de *OOCSMP* (ver la figura VIII.1), ya que sus objetivos son distintos, sin embargo son parte del mismo lenguaje (ambos tipos de instrucciones se compilan con el mismo compilador, *C-OOL*).

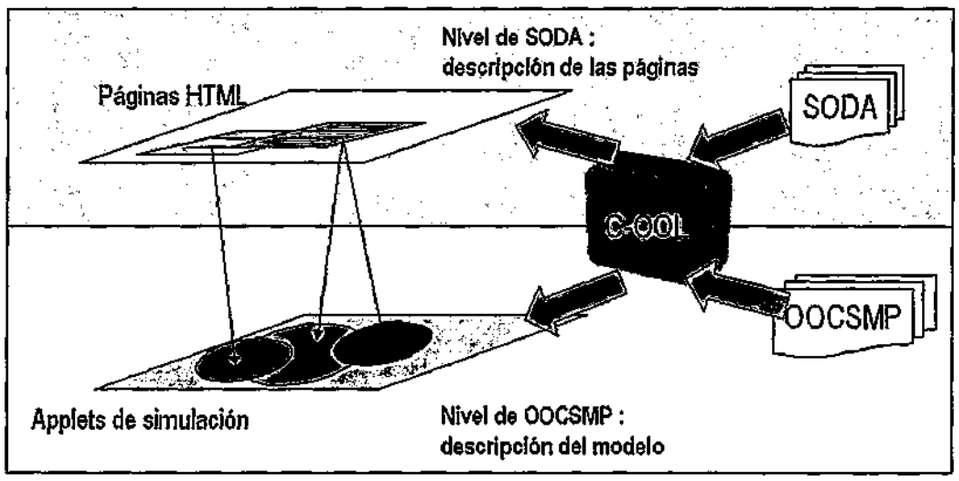


Figura VIII.1: Niveles *SODA* y *OOCSMP*

Las instrucciones de *SODA* no son un mero disfraz de la sintaxis de *HTML*, sino que desde *SODA* se puede acceder al nivel del modelo definido por *OOCSMP*. Por ejemplo, se puede acceder al valor inicial de las variables del modelo de simulación, así como evaluar expresiones que involucren a variables 'especiales' o a variables del modelo.

Además, el uso de *SODA* tiene otras ventajas respecto a la construcción directa de las páginas del curso con un editor *HTML*:

- Es posible la reutilización de código *SODA* (mediante la instrucción *INCLUDE*) entre las páginas del curso, por ejemplo, el código que define el índice del curso, datos sobre los autores, etc. Esto además va a mejorar la uniformidad de apariencia de las páginas del curso.
- Se han aumentado los elementos disponibles directamente desde la sintaxis *HTML* (que dispone solamente de imágenes), añadiendo gráficos 3D y 2D de funciones (que se introducen como expresiones *OOCSMP*).
- Hay variables *SODA* "especiales", tales como:
  - Contadores de figuras, tablas, gráficos, modelos y elementos de listas.
  - Variables que indican la fecha actual, el fichero que se está compilando, el autor, su correo electrónico, etc.

Este último grupo de variables nos va a permitir una mayor generalidad en la definición de las páginas, ya que no es necesario, por ejemplo, referirse al nombre explícito del autor de la página, o a su dirección de correo. Estos datos se definen una sola vez en estas variables, o en un fichero aparte. Más

tarde, si cambia, no se ha de modificar cada página del curso, sólo recompilarlas. De esta forma es posible crear índices o pies de página genéricos. Al incluirse estos ficheros en el archivo que estamos compilando, estas variables se instanciarán.

- Es posible definir estilos compuestos de escritura (similares a los de cualquier procesador de textos).
- Se pueden definir macros que indican cómo se ha de traducir ciertos "patrones". Mediante esta última posibilidad, en teoría sería posible configurar el sistema para generar otro tipo de documentos distintos de *HTML*, tales como *LaTeX*.

También se pueden crear tablas, links, etc., e incluir directamente sintaxis *HTML* dentro del código *SODA*.

VIII.1.1 Hacia una herramienta de autor para la creación de cursos basados en simulación.

Las herramientas de autor (*authoring tools*) engloban herramientas que permiten al usuario diseñar el aspecto de cierto tipo de aplicaciones, formadas por varias 'pantallas'. Normalmente las aplicaciones que se diseñan con herramientas de autor tienen componentes multimedia, y diversos *widgets*, que este tipo de herramientas ayuda a colocar y sincronizar. También se puede configurar la navegación entre las distintas pantallas de la aplicación. Con estas herramientas se diseñan cursos y presentaciones multimedia, páginas web, etc. El ejemplo típico de herramienta de autor para el diseño de presentaciones multimedia es Macromedia Director[MACR00],

Las herramientas de autor para la web (*Web authoring tools*) permiten el diseño de las páginas de un sitio Web [Webe95]. Los entornos van desde los enteramente visuales, *WYSIWYG* puros, en los que no es necesario un conocimiento de *HTML*, hasta los no visuales basados en mandatos [Mend00]. Algunas herramientas de autor para la web son:

- Drumbeat [Drum00], NetObjects Fusion[Neto00], que son enteramente visuales.
- Macromedia Dreamweaber1.0 [Drea00], HoTMetal PRO 4.0 [HotM00], Microsoft FrontPage98 [Fron00] o Grif [Quin95] en los que se ofrece un enfoque mixto, se pueden hacer la mayoría de tareas en un editor *WYSIWYG*, pero se puede ver y modificar el código *HTML*.
- HotDog Professional 4 [HotD00], HomeSite [Home00], etc. son enteramente textuales, se trabaja directamente con tags *HTML*, si bien contienen 'guías' (*wizards*) para crear tablas, frames etc.

Además Dreamweaber ofrece una extensión (*course builder*) para la creación de cursos de enseñanza. Estas extensiones están orientadas a la creación de páginas de tests, en las que se hace un seguimiento del alumno y se recogen repuestas a las preguntas. Además el profesor puede recopilar fácilmente datos estadísticos sobre las interacciones de cada alumno con el curso. Las páginas creadas con Dreamweaber no contienen *applets* ni necesitan plug-ins, los contenidos son *HTML*, *DHTML* o JavaScript.

Ninguna de las herramientas de autor para la web mencionadas satisface nuestras necesidades de una herramienta en la que se puedan diseñar las simulaciones y las páginas del curso, de modo que sea posible un buen acoplamiento de ambos.

El nivel definido por *SODA* nos va a permitir acercarnos a la idea de herramienta de autor para la creación de cursos basados en simulación para Internet. Nuestro enfoque se aproxima por el momento al de las herramientas textuales, ya que *C-OOL* no dispone de entorno gráfico. Esta es una de las líneas abiertas que se expone en la sección de trabajo futuro.

■ VIII.2 *SODA*

Las instrucciones conocidas como *SODA* se hicieron necesarias, como ya se ha comentado, debido a la idea de construir cursos para Internet que contuvieran los *applets* de simulación generados con *C-OOL*. Aunque las instrucciones *SODA* se pueden ver como de un nivel más alto que el resto de *OOCSMP*, desde *SODA* se pueden acceder al nivel *OOCSMP*, para obtener el valor inicial de las variables de los modelos de simulación, crear gráficos 2D y 3D de expresiones *OOCSMP*, etc. Además es posible mezclar en un mismo fichero instrucciones *SODA* y *OOCSMP*, si bien las instrucciones *SODA* irán a parar a una página *HTML*, y las instrucciones *OOCSMP* que definen el modelo a un fichero Java.

Para generar una página *HTML* mediante *C-OOL*, es necesario compilar con la opción *-HTMLpage=<pagina>* (ver sección IX).

A continuación se describen las instrucciones *SODA*.

■ VIII.2.1 Creación de enlaces, inserción de imágenes, barras de separación y tablas

Desde *SODA* es posible definir enlaces, insertar imágenes y tablas, con sintaxis similar a la de *HTML*:

```
LINK    [[posicion],] "pagina", "descripción"
IMAGE   [[posicion],] "imagen" [,[posicion]], "titulo de la imagen"
BAR     [[posicion[,porcentaje]]]
TABLE   ["titulo de la tabla",][<dimx>;<dimy>],
        [[posicion[,porcentaje]]] (, "celda-n")*
        [, "titulo de la tabla"]
```

Sintaxis VIII.1: Instrucciones *SODA* para la creación de enlaces, inserción de imágenes y tablas.

La forma de alinear cada uno de los elementos, se especifica en el argumento *posicion*, y puede ser alineado a la izquierda (*E*), centrado (*C*) o a la derecha (*W*).

- En la instrucción *IMAGE*, se puede especificar además un título para la imagen, que puede ir encima (*N*) o debajo (*S*) de la imagen.
- La instrucción *BAR* crea una línea de separación horizontal, que ocupa un cierto porcentaje (parámetro *porcentaje*) de la página.
- En la instrucción *TABLE*, si se especifica el primer parámetro (*"titulo de la tabla"*) el título se colocará encima de la tabla, mientras que si se especifica el último, se colocará debajo. Además se puede especificar el porcentaje de la página que ocupará la tabla (parámetro *porcentaje*). Los elementos de la tabla se especifican entre comillas y separados por comas.

Los elementos de la tabla y en general cualquier texto descriptivo, pueden contener las instrucciones *SODA* que comienzan por '\', por ejemplo, variables *SODA*, gráficos 2 y 3D, instrucciones para el formateo de texto, etc.

Además, *C-OOL* tiene un contador de las tablas y de las imágenes que han sido insertadas hasta el momento. Estos contadores se presentan en el apartado VIII.2.3.

■ VIII.2.2 Títulos y descripciones textuales

La primera de ellas, ya se ha comentado con anterioridad, es la instrucción *TITLE*. Esta instrucción, además de crear un título en algunos de los gráficos, también crea el título de la página *HTML* y de la ventana del navegador.

La instrucción *DESCRIPTION* admite una línea de texto como parámetro, que será traducida a *HTML* (acentos, caracteres especiales, etc.) y puesta como un párrafo. Si hay más de una instrucción *DESCRIPTION* seguida, su texto se junta con el párrafo anterior. En el texto de una instrucción *DESCRIPTION* se pueden insertar caracteres de salto de línea (mediante el carácter especial '\n') y en general cualquier instrucción *SODA* que comience por '\'. Además, como primer parámetro opcional de *DESCRIPTION*, se puede indicar si queremos centrar el texto, o lo queremos alineado a la derecha (mismo formato que *LINK* e *IMAGE*). La sintaxis de esta instrucción queda como sigue:

```
DESCRIPTION [[position],] texto
```

Sintaxis VIII.2: Sintaxis de la instrucción *DESCRIPTION*

■ VIII.2.3 Variables *SODA* y contadores

SODA dispone de una serie de variables para identificar al autor de la página, su correo electrónico, la fecha de creación, etc. Estas variables se inicializan de la siguiente forma:

```
AUTHOR <nombre-del-autor>
DATE <fecha-de-creación>
EMAIL <dirección-de-correo-electrónico>
```

Sintaxis VIII.3: Inicialización de variables *SODA*.

Suele ser útil inicializar dichas variables en un fichero separado, que será común a todas las páginas del curso (mediante la instrucción *INCLUDE*), de forma que si alguno de estos datos cambiase, sólo se ha de modificar un fichero.

El acceso a estas variables se puede realizar en cualquier instrucción *SODA*, mediante las sintaxis:

```
\AUTHOR
\EMAIL
\DATE
```

Sintaxis VIII.4: Acceso al nombre del autor, e-mail y fecha.

Una referencia a la variable *EMAIL*, crea el tag *HTML* "mailto:", de forma que al pinchar en él, se abre una ventana para mandar un mensaje a la correspondiente dirección. También existen variables (*\CURR_DATE* y *\CURRFILE*) que proporcionan la fecha actual y el fichero que se está compilando.

Estas variables permiten crear índices y pies de páginas genéricos, con información sobre el autor, la fecha de creación del modelo, etc. Estos ficheros se especializarán cuando se incluyan en un fichero *SODA* en el que hayan tomado valor estas variables.

En general, en cualquier instrucción *SODA*, se puede acceder a los contadores (de imágenes, de tablas, de elementos de lista, de gráficos 2D, 3D, mapas de isosuperficies y modelos), mediante las sintaxis:

```
\ICOUNT          \ITEMCOUNT
\TCOUNT          \2DCOUNT
\3DCOUNT         \ISOCOUNT
\MCOUNT
```

Sintaxis VIII.5: Contadores *SODA*.

Los contadores se incrementan después de insertar un elemento del tipo correspondiente. El contador de elementos cuenta los elementos que se han insertado en una tabla en el nivel actual.

■ VIII.2.4 Estilos de escritura

La definición de estilos de escritura se realiza mediante las instrucciones:

```
\LINK{ "hipervinculo" texto }
\TARGET{...}
\ITALIC{...}
\BOLD{...}
\SUP{...}
\SUB{...}
\CENTER{...}
\LEFT{...}
\RIGHT{...}
\SIZE= <size>{...}
\FONT="font"{...}
```

Sintaxis VIII.6: Estilos de escritura.

El primero (*LINK*) sirve para generar un hipervínculo a la página *"hipervinculo"*, dentro de *texto* se puede encontrar cualquier variable especial o estilo.

El segundo (*TARGET*) sirve para insertar una marca dentro de una página, que pueda ser referenciada por un hipervínculo. Entre las llaves debe ir el nombre de la marca.

Los dos siguientes (*ITALIC* y *BOLD*) sirven para escribir en itálica y en negrita. *SUP* y *SUB* se usan para escribir en superíndice y en subíndice. *CENTER*, *LEFT* y *RIGHT* para centrar el texto, alinearlo a la izquierda o a la derecha respectivamente. Los dos últimos (*SIZE* y *FONT*) para especificar el tamaño de la letra en relación con el tamaño actual (p.e. si el parámetro *<size>* vale 2, se aumenta en dos puntos el tamaño, si vale –2, se disminuye en dos el tamaño), y el tipo de letra. En todos los casos, se ha de cerrar la llave cuando termine el estilo.

Es posible también declarar listas no numeradas mediante la instrucción *\ITEM{*. No se ha de especificar cuándo empieza o termina la lista, el compilador lo deduce por el contexto. Se pueden anidar listas, especificando una instrucción *\ITEM{* dentro de otra. Si se quieren listas anidadas, basta con incluir el contador de elementos de lista *\ITEMCOUNT* detrás de *\ITEM*, si bien es más cómodo especificar un estilo compuesto, como se mostrará en el apartado VIII.2.5

■ VIII.2.5 Estilos compuestos

La instrucción *STYLE* permite definir estilos compuestos y tiene como sintaxis:

```
STYLE "nombre", "estilo-1 (estilo-n )*"
```

Sintaxis VIII.7: Declaración de estilos compuestos.

Donde *nombre* es el nombre del estilo compuesto, y *estilo-1... estilo-n* son los estilos de los que está compuesto el estilo *nombre*, que pueden ser a su vez un estilo compuesto.

Por ejemplo, si queremos definir un estilo para crear una lista numerada que tenga el número en negrita, cursiva y un punto mayor, lo podríamos definir del siguiente modo:

```
STYLE "\NITEM{", "\ITEM{ \ITEMCOUNT \BOLD{ \ITALIC{"
```

Ejemplo VIII.1: Estilo para listas numeradas (en fichero *styles.csm*).

Es obligatorio que el nombre de un estilo compuesto empiece por '\', aunque no lo es que termine en una llave, pero es una regla mnemotécnica, que puede ayudar a recordar que hay que cerrar el estilo con una llave cerrada. Es conveniente guardar los estilos en fichero separado para su posterior reutilización. Hemos desarrollado una biblioteca de estilos básicos (fichero *styles.csm*), que se ha usado en la construcción de los cursos que se detallan en el capítulo X. El estilo del ejemplo anterior (VIII.1) se podría utilizar de la siguiente forma:

```
INCLUDE "styles.csm"
DESCRIPTION Se han generado los siguientes cursos:\n
DESCRIPTION \NITEM{Gravitación.}
DESCRIPTION \NITEM{Ecología.}
DESCRIPTION \NITEM{Electrónica.}
DESCRIPTION \NITEM{PDEs.}
```

Ejemplo VIII.2: Ejemplo de uso del estilo para listas numeradas.

En el ejemplo anterior (VIII.2) vemos que estamos reutilizando los estilos definidos en `styles.csm`, y que no nos ha hecho falta indicar el principio o el fin de las listas.

■ VIII.2.6 Acceso al nivel *OOCSMP*

En los siguientes dos apartados, se presentan instrucciones *SODA* que permiten el acceso a información de nivel de descripción del modelo de simulación.

VIII.2.6.1 Inclusión de modelos de simulación en la página

La instrucción *MODEL*, cuya sintaxis se muestra en VIII.9, genera el tag *HTML* necesario para incluir el modelo *"modelo"* en la página. Los parámetros *height* y *width* indican el alto y el ancho del *applet*. Se puede indicar si se quiere centrado (*[C]*), a la izquierda (*[E]*), o a la derecha (*[W]*). Se supone que el modelo *"modelo"* es un programa *OOCSMP* que se ha compilado previamente. El último parámetro es opcional, y se corresponde con el parámetro *codebase* del tag *HTML*. Este parámetro especifica el directorio que contiene el *applet*. Por defecto es el directorio que ocupe el fichero *HTML*.

```
MODEL [height;width], [position], "modelo" [, "codebase"]
```

Sintaxis VIII.9: Sintaxis de la instrucción *MODEL*

VIII.2.6.2 Acceso a variables del modelo, expresiones

Se puede acceder al valor inicial de las variables del modelo, con sintaxis similar a las variables de la sección VII.2.3:

```
\<nombre-variable>
```

Sintaxis VIII.10: Acceso a variables del modelo.

Se puede también acceder al valor inicial de los atributos de un objeto. Si la variable es una matriz, un vector, o una expresión del tipo *<clase>.<atributo>* o *<colecc-objetos>.<atributo>* se crea una tabla en la que se muestra el valor inicial.

Para evaluar una expresión, en la que pueden intervenir todas las variables ya comentadas, se ha de encerrar la expresión entre dos símbolos *'$'*. En la expresión sólo pueden aparecer operadores simples, no bloques *OOCSMP*. También pueden aparecer variables del modelo. Un ejemplo sencillo de uso de expresiones se muestra en el siguiente fragmento de código:

```
...
DESCRIPTION La figura $\ICOUNT+1$ muestra el esquema del problema
IMAGE [C], "problem.jpg", "Figura \ICOUNT: Esquema del problema"
...
```

Ejemplo VIII.3: Ejemplo de uso de expresiones.

■ VIII.2.7 Macros de traducción, inclusión de código *HTML*

Con las macros de traducción, es posible indicar al compilador cómo tiene que traducir ciertos patrones. Esta es una instrucción potente, ya que teóricamente se podrían crear macros para que el compilador generara otro tipo de lenguaje basado en etiquetas en vez del *HTML*, como por ejemplo *LaTeX*.

Las macros de traducción se declaran de la siguiente forma:

```
TRANSLATE "nombre ( param-1 )*", "patrón"
```

Sintaxis VIII.11: Declaración de estilos compuestos.

Donde nombre es el nombre de la macro, *param-1... param-n* son los parámetros de la macro, y *patrón* es un patrón donde se le indica al compilador cómo ha de traducir la macro. Normalmente el patrón incluirá instrucciones *HTML*, y además, puede incluir el nombre de los parámetros formales de la macro, en cuyo caso, cuando se invoque a la macro, se sustituirán por los parámetros actuales. El patrón también puede contener variables especiales. Cuando se invoca una macro, los parámetros han de ir entre comillas dobles, y separados por comas.

Por ejemplo, el siguiente es un ejemplo muy sencillo de declaración de un par de macros para generar títulos de secciones:

```
TRANSLATE "SECTION a", "<H1> \BOLD{a} \TARGET{a}<\H1>"
TRANSLATE "SECTIONn n a", "<Hn> \BOLD{a} \TARGET{a}<\Hn>"
```

Ejemplo VIII.4: Ejemplo de declaración de macros para secciones (fichero *macros.csm*).

La primera de las dos macros anteriores tiene como propósito generar un título de primer orden con el texto en negrita. La segunda macro sirve para generar títulos de orden variable (parámetro *n* de la macro). Estas dos macros crean además una marca por si se quiere referenciar el título mediante un hipervínculo. Se ha desarrollado una biblioteca de macros (fichero *macros.csm*), que ha sido usada en la construcción de los cursos que se detallan en el capítulo X. Las dos macros anteriores se utilizarían de la siguiente forma:

```
INCLUDE "macros.csm"
SECTIONn "2", "Esto es una sección de orden dos"
```

Ejemplo VIII.5: Ejemplo de uso de las macros para secciones.

También se puede incluir código *HTML* en la definición de las páginas mediante la instrucción:

```
HTML <código-HTML>
```

Sintaxis VIII.12: Instrucción para la inserción de código *HTML*.

■ VIII.2.8 Gráficos 2D, 3D y mapas de isosuperficies.

Es posible insertar gráficos en dos y tres dimensiones y mapas de isosuperficies estáticos en las páginas del curso. Estos *widgets* se han implementado reutilizando las bibliotecas Java que son usadas por los *applets* de simulación.

La sintaxis es la siguiente:

```
3DGRAPHIC[posicion], [tamX; tamY], [filas;cols],
          [minX, maxX, minY, maxY], expresion [, "codebase"]

2DGRAPHIC[posicion], [tamX; tamY], [cols],
          [minX, maxX], expresion [, "codebase"]
```

```
ISOSURFACE [posicion], [tamX; tamY], [filas;cols],
          [minX, maxX, minY, maxY], expresion [, "codebase"]
```

Sintaxis VIII.13: Sintaxis de los gráficos 2D, 3D y mapas de isosuperficies.

Al igual que en las instrucciones anteriores, el parámetro *posicion* puede ser *C*, *E* o *W*, e indica si se quiere el elemento centrado, a la izquierda o a la derecha. Los parámetros *tamX* y *tamY*, indican el tamaño en pixeles que se quiere para el elemento. *filas* y *cols* indican el número de filas y columnas que ha de tener la muestra que se tome de la expresión a representar. La escala *X* e *Y* se ha de indicar en los parámetros *minX*, *maxX*, *minY* y *maxY*. *expresion* puede ser cualquier expresión *OOCSMP*, y finalmente "*codebase*" es un parámetro opcional y tiene el mismo significado que el de la instrucción *MODEL* (sección VIII.2.6.1).

Además, existe una versión con idéntica sintaxis de estas tres instrucciones con un '\' delante, de forma que se pueda incluir en cualquier descripción textual, dentro de tablas, etc.

■ VIII.3 Ejemplos

Con todas estas instrucciones, se reduce drásticamente el esfuerzo de retocar las páginas *HTML* que se generaban, cosa que había que hacer en anteriores versiones de *C-OOL*. Además, si el compilador detecta que no se ha declarado ningún modelo, sino que sólo hay información para la página *HTML* - para, por ejemplo generar la página principal del curso con el índice - no se genera código Java, si no sólo páginas *HTML*.

En los siguientes listados se muestra un esquema del código *SODA* necesario para generar dos páginas del curso de *PDEs*. La primera es una página de teoría, la introducción al método de los elementos finitos. La segunda es una página de aplicaciones, donde se muestra el problema del calentamiento de las piezas móviles.

```
* AUTHOR Juan de Lara
* EMAIL Juan.Lara@ii.uam.es
* DATE 15/1/2000
TITLE The Finite Element Method
DESCRIPTION The method consists in aproximating the function in small
DESCRIPTION domain portions called finite elements, or simplex....
...
TABLE [1;2], [C,80],
       "\2DGRAPHIC [C], [200;200], [2], [0, 1], 1-X,"\BOLD{N\SUB{1,1}(x)}"",
       "\2DGRAPHIC [C], [200;200], [2], [0, 1], X,"\BOLD{N\SUB{2,1}(x)}""
       "1-D Linear Shape Functions"
TABLE [1;3], [C,80],
      "\2DGRAPHIC [C],[200;200],[25],[-1, 1],0.5*(X*X-X),"\BOLD{N\SUB{1,2}(x)}"",
      "\2DGRAPHIC [C], [200;200], [25], [-1, 1], -(X+1)*(X-1),"\BOLD{N\SUB{2,2}(x)}"",
      "\2DGRAPHIC [C], [200;200], [25], [-1, 1], 0.5*(X+1)*X,"\BOLD{N\SUB{3,2}(x)}"",
      "1-D Quadratic Shape Functions"
...
INCLUDE "pdes\pdeindex.csm"
INCLUDE "coursesindex.csm"
INCLUDE "footnote1.csm"
```

Listado VIII.1: Esquema del código *SODA* necesario para la generación de una página introductoria al método de los elementos finitos.

En esta página hemos insertado cinco gráficos bidimensionales (las funciones de forma lineales y cuadráticas 1-d) dentro de dos tablas, y además, las tres últimas líneas reutilizan archivos que describen índices del curso de *PDEs* (común a todas las páginas de dicho curso), un índice al resto de los cursos (común a todas las páginas de todos los cursos) y un pie de página con información sobre la creación de la página (común a todas las páginas de todos los cursos). Un esquema de estos ficheros se muestra a continuación:

```
BAR [C,75]
DESCRIPTION \SIZE=-2{ Last modified \DATE by \AUTHOR (\EMAIL,
DESCRIPTION \LINK{"http://www.ii.uam.es/~jlara" http://www.ii.uam.es/~jlara})
DESCRIPTION need \LINK{"../help.html"help} for using this courses?.}
```

Listado VIII.2: Fichero *footnote1.csm*: pie de página común a todas las páginas del curso.

El fichero anterior supone que se han definido las variables *AUTHOR*, *DATE* y *EMAIL* en el archivo en el que se incluye. De esta forma se obtiene un pie de página general, usable con datos distintos.

```
INCLUDE "styles.csm"
BAR [C,75]
TABLE "Theory pages:", [1;4], [C,80],
 "\CLINK{"pdes.html"Main page}",
 "\CLINK{"FEM.html"FEM (i)}",
 ...
TABLE "Example pages:", [4;2], [C,80],
 "\CLINK{"Heat.html"1-d Heat Equation}",
 "\CLINK{"h2d.html"2-d steady state Heat Equation}",
 ...
TABLE "Application pages:", [5;1], [C,80],
 "\CLINK{"LBeam1.html"Heating of two beams}",
 "\CLINK{"LBeam1.html"Heating of two moving beams}",
 ...
```

Listado VIII.3: Esquema del fichero *pdeindex.csm*: índice para el curso de *PDEs*, común a todas las páginas de dicho curso.

En el fichero anterior hemos colocado enlaces a las diversas páginas del curso en tres tablas. El estilo *\CLINK* es un estilo compuesto definido como *STYLE "\CLINK{", "\CENTER{ \LINK{"*, y que se incluye en la biblioteca de estilos "*styles.csm*". El archivo *coursesindex.csm*, común a todas las páginas del curso es similar al fichero *pdeindex.csm*, pero con enlaces a la página principal de cada curso. La figura VIII.2 muestra el resultado de la compilación del código del listado VIII.1

El siguiente listado muestra un esquema de las instrucciones necesarias para generar una página *HTML* con el modelo de las dos piezas móviles:

```
* AUTHOR Juan de Lara
* EMAIL Juan.Lara@ii.uam.es
* DATE 15/1/2000
TITLE Heating of two moving pieces
DESCRIPTION Suposse we want to simulate the heating of two long L-shaped
...
IMAGE [C], "../images/proble2.gif" , "Figure \ICOUNT : Scheme of the Problem"
DESCRIPTION The heat supplied follows the equation :
DESCRIPTION \ITALIC{e\SUP{2t}sin(x+y)cosh(x+y)}. The following graphic
DESCRIPTION shows the heatat time = 0.\n
3DGRAPHIC [300;300], [C], [15;15], [0, 5, -1, 1], SIN(X+Y)*CH(X+Y),
        "Heat supplied at TIME=0", "/tesis/courses"
...
DESCRIPTION The following applet simulates the described problem. The problem
DESCRIPTION is solved using the finite element method.
MODEL [500;500], [C], "p4", "\tesis\courses"
INCLUDE "pdes\pdeindex.csm"
INCLUDE "coursesindex.csm"
INCLUDE "footnote1.csm"
```

Listado VIII.4: Esquema del código *SODA* necesario para la generación de la página del modelo del calentamiento de las piezas móviles.

En el listado anterior, hemos incluido una imagen con la descripción del problema, un gráfico tridimensional y el *applet* de simulación. Además estamos reutilizando los índices *pdeindex.csm*, *coursesindex.csm* y *footnote1.csm*. El resultado de compilar el listado anterior se muestra en la figura VIII.3.

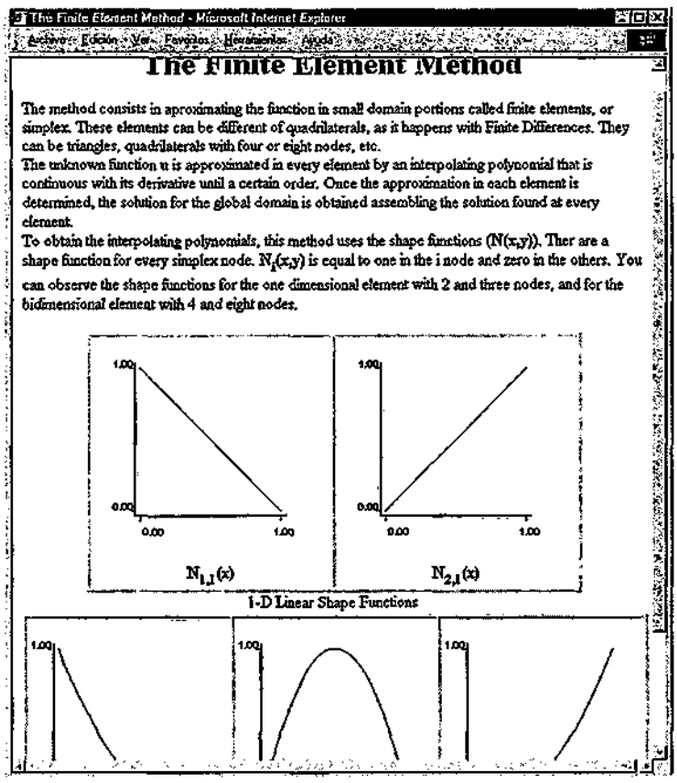


Figura VIII.2: Página de introducción al método de los elementos finitos.

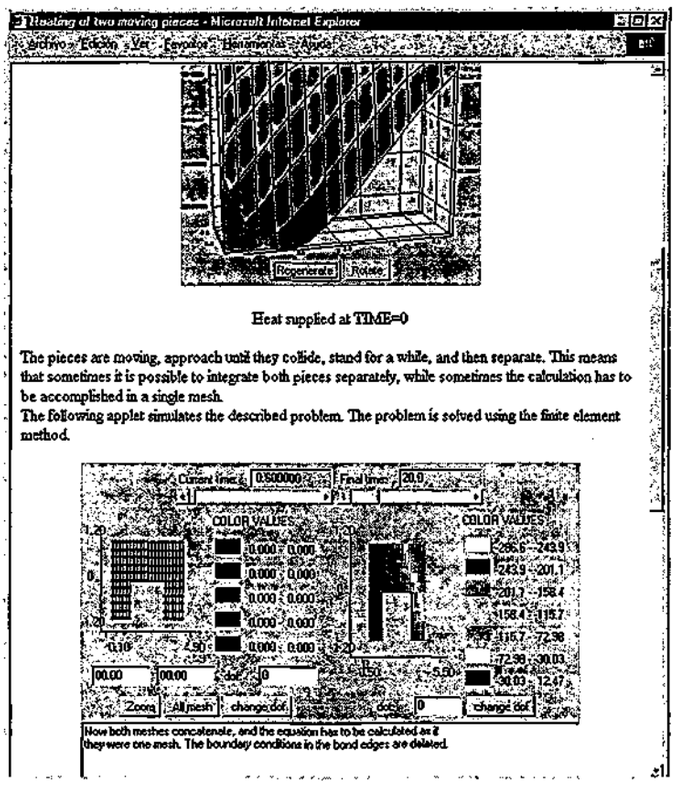


Figura VIII.3: Página con el modelo del calentamiento de las piezas móviles.

■ VIII.4 Generación automática de documentación

Se ha ampliado la instrucción de comentarios ('*'), para facilitar la generación automática de ficheros *HTML* con documentación del modelo. Esto se hace con la opción *-genDoc* del compilador *C-OOL* (ver capítulo IX).

Es decir, es posible insertar comentarios especiales, con información adicional al modelo, que crean secciones especiales en los ficheros de documentación, o modifican la apariencia visual de la documentación. También se puede insertar código *HTML* directamente. Muchas de las instrucciones de comentario coinciden con las expuestas en *SODA*, pero dentro de la instrucción de comentario, éstas son:

```
* TITLE <titulo del modelo>
* AUTHOR <nombre del autor>
* EMAIL <dirección e-mail>
* DATE <fecha>
* ABSTRACT <resumen>
* BAR    [[posicion [,porcentaje]]]
* HTML   <código HTML>
* LINK   [[posicion],] "pagina", "descripción"
* IMAGE  [[posición],] "imagen" [,[posicion]], "titulo de la imagen"
* TABLE  ["titulo de la tabla",][<dimx>;<dimy>],
         [[position[,parametro]]] (, "celda-n")*
         [, "titulo de la tabla"]
** <comentario>
```

Sintaxis VIII.14: Comentarios especiales.

En todas estas instrucciones, el texto que lleven como parámetro se analiza, tratándose convenientemente símbolos o variables especiales, expresiones o variables del modelo.

- *TITLE*, *AUTHOR*, *DATE* y *ABSTRACT* crean secciones especiales para el título, el autor, la fecha de creación del modelo o componente y un resumen de las características del modelo. Se puede incluir una sentencia de cada tipo por cada fichero *OOCSMP*.

- Si en la línea posterior a un comentario de tipo *TITLE*, *AUTHOR*, *DATE* o *ABSTRACT*, aparece otro comentario que empiece por '**' se considera que esa línea pertenece a la misma sección de la documentación.

El resto de las instrucciones de comentario es equivalente a la correspondiente instrucción *SODA*, pero actuando sobre las páginas *HTML* de la documentación.

El siguiente ejemplo muestra un modelo, compuesto por dos ficheros (VIII.5, *grav.csm* y VIII.6, *Planet.csm*), en el que se han insertado varios de estos comentarios especiales, las figuras VIII.2 y VIII.3 muestran las páginas *HTML* tal y como se verían en un navegador.

```
TITLE INNER SYSTEM
* HTML <BODY BACKGROUND="ck040bg.gif">
* TITLE    Inner system simulation
* ABSTRACT This is the simulation of the motion of the inner system planets.
** The way it is done is encapsulating the behaviour of a generic planet
** inside a class called Planet, and declaring an object per each planet
** in the system.
* AUTHOR Juan de Lara
* EMAIL Juan.Lara@ii.uam.es
* DATE    2/11/99
* BAR
DATA G:=0.00011869, PI:=3.141592653589793
* We are declaring the G and Pi constants
DATA MS:=332999
* The sun data (MS)
```

```
INCLUDE "planet.csm"
* Declare actual planets
Planet Mercury("Mercu",0.055271,-0.3871, 0,      2.078, -9.892, 7.004)
Planet Venus  ("Venus",0.81476,  0.7233, 0,      0.051,  7.39,  3.394)
Planet Earth  ("Earth",1,        0,      1,     -6.2899, 0.107, 0    )
Planet Moon   ("Moon", 0.01235,  0,      0.9975,-6.0783, 0.107, 0    )
Planet Mars   ("Mars", 0.10734,  1.5233, 0,      0.476,  5.071, 1.85 )
Planet Apollo ("Apolo",1957E-14, 0,      1.4849,-4.253,  2.915, 6.4  )
Planet Jupiter("Jupit",317.94,   0,     -5.2028, 2.754,  0.131, 1.308)
Planet InnerSystem := Mercury, Venus, Earth, Moon, Mars, Apollo, Jupiter
InnerSystem.STEP()
InnerSystem.ACTION(InnerSystem)
* Time intervals and other data
TIMER delta:=.0005, FINTIM:=2, PRdelta:=.1, PLdelta:=.01
METHOD ADAMS
* LINK "grav.csm", "Complete OOCSMP listing"
* LINK "Planet.csm", "Planet class definition"
```


Listado VIII.5: Ejemplo de código con comentarios especiales, fichero *grav.csm*

```
* TITLE    File containing class planet
* HTML <BODY BACKGROUND="ck040bg.gif">
* ABSTRACT This file contains the planet class, used to encapsulate
** all the behaviour of a planet
* AUTHOR   Juan de Lara
* EMAIL Juan.Lara@ii.uam.es
* DATE     2/11/99
CLASS Planet (
 * Definition of Planet class
 NAME name
 * Name of the planet (name)
 DATA M, X0, Y0, XP0, YP0, FI
 * Mass (M), initial position (X0,Y0), initial velocity (XP0, YP0), FI
 INITIAL
    FIR:=FI*PI/180
    CFI:=COS(FIR)
    SFI:=SIN(FIR)
    * Compute initial data (FIR, CFI and SFI)
 DYNAMIC
    * Distance to the Sun
    R2   := X*X+Y*Y
    R    := SQRT(R2)
    Y1   := Y*CFI
    Z    := Y*SFI
    * Mutual influences
    * The Sun on this planet
    APS := G*MS/R2/R
    * This planet on the Sun
    ASP := G*M/R2/R
    XPP := -(ASP+APS)*X
    YPP := -(ASP+APS)*Y
    XP  := INTGRL(XP0,XPP)
    YP  := INTGRL(YP0,YPP)
    X   := INTGRL(X0,XP)
    Y   := INTGRL(Y0,YP)
 ACTION Planet P
    * Mutual actions of two planets
    * Distance to another planet
    DPP2 := (P.X-X)*(P.X-X)+(P.Y-Y)*(P.Y-Y)+(P.Z-Z)*(P.Z-Z)
    DPP  := SQRT(DPP2)
    * Influences
    * The other planet on the Sun
    ASP1 := G*P.M/P.R2/P.R
    * The other planet on this planet
    APP1 := G*P.M/DPP2/DPP
    * Coordinate conversion
    Y2 := P.Y*COS(P.FIR-FIR)
    * Actual action of the planet
```

```
    XPP += APP1*(P.X-X) - ASP1*P.X
    YPP += APP1*(Y2-Y) - ASP1*Y2
PLOT Y,X
FINISH R<=.005
}
```

Listado VIII.6: Ejemplo de código con comentarios especiales, fichero *Planet.csm*

El resultado de compilar el programa `grav.csm` anterior mediante la opción `-genDoc` es un programa Java con la simulación del sistema solar interior (ver capítulo X), y las dos siguientes páginas *HTML* (figuras VIII.4 y VIII.5).

Como puede advertirse en estas dos páginas, se aprovecha la información que contiene la tabla de símbolos durante la compilación del modelo para generar documentación sobre, por ejemplo, los objetos que se crean y su clase (se crea un enlace a la página donde se haya definido la clase), los parámetros que tiene un método, la clase de estos, etc.

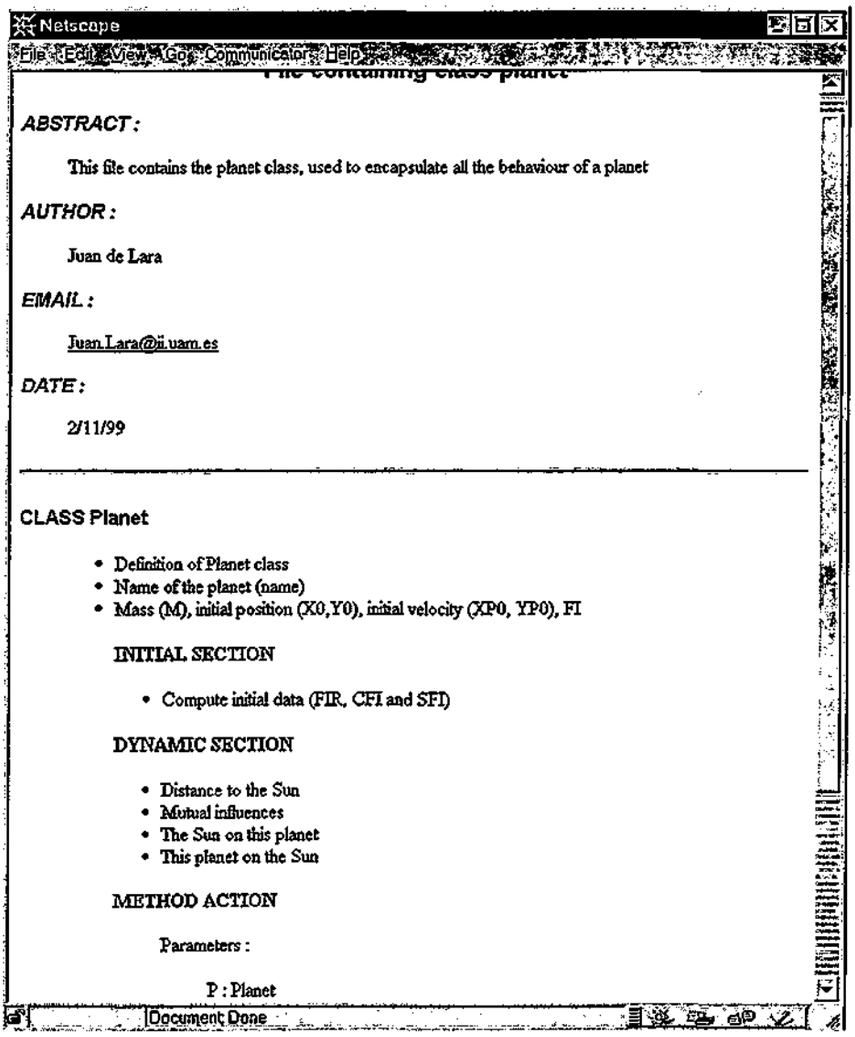


Figura VIII.4: Documentación generada automáticamente, *docPlanet.html*

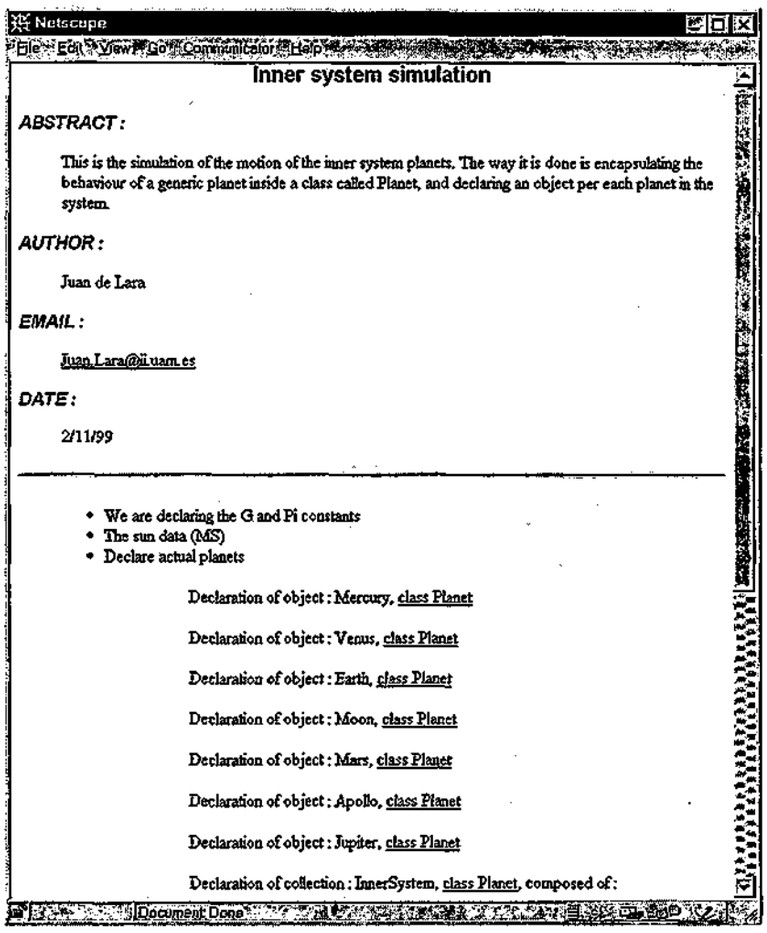


Figura VIII.5: Documentación generada automáticamente, *docgrav.html*

IX. El compilador *C-OOL*

En este capítulo se presenta el compilador *C-OOL* (a Compiler for the OOcsmp Language) que hemos desarrollado para compilar el nuevo lenguaje *OOCSMP*. El capítulo se divide en los siguientes apartados:

- La sección 1 presenta una introducción, en la que se describe un esquema general del funcionamiento de *C-OOL*. En este apartado se describen brevemente todos las opciones que admite el compilador.
- La sección 2 presenta el funcionamiento del compilador y de la interfaz generada, en el caso de que se genere código *C++* para *DOS*. También se explica cómo compilar los archivos *C++* generados por el compilador.
- La sección 3 muestra la interfaz de usuario cuando se genera código *C++* para *Amulet* [Myer97] [Open97]. En esta sección también se muestra un ejemplo de generación de modelos de tareas y escenarios, y se propone una arquitectura conjunta con *ATOMS* [Garc98] [Fede99] [Rodr97] para la generación automática de interfaces de usuario inteligentes.
- La sección 4 presenta el funcionamiento de la interfaz cuando se decide generar código Java sin distribución. También se explica brevemente cómo compilar los programas Java generados, y cuál es la estructura del código Java generado, así como las diferencias y dificultadas encontradas en diseñar un compilador que sea capaz de generar diversos lenguajes objeto.
- La sección 5 presenta una característica importante de la interfaz generada, tanto para Java como para Amulet: permite la inserción, el borrado o el cambio de colección de objetos durante la ejecución.
- En la sección 6 se explican detalles de cómo compilar una simulación distribuida.

IX.1 Introducción

C-OOL es un compilador en línea que permite compilar programas escritos en el lenguaje *OOCSMP* y en su antecesor, *CSMP-III*. Además, con la compilación se genera una interfaz, para que el usuario interaccione con el problema de simulación. *C-OOL* está programado en C++, y funciona bajo Windows'95.

C-OOL es capaz de generar código C++ [Stro97] para *DOS* o bien C++ que use la librería *Amulet* [Myer97]) o Java [Java99] (antiguo o nuevo modelo de eventos), por tanto este código puede ser portado a casi cualquier plataforma, entre ellas *DOS*, *Windows95*, *Unix*, *Macintosh*, etc.

También es capaz de generar código de simulación distribuido para varias máquinas, mediante el uso de los paquetes *RMI* (Remote Method Invocation) [Berg97] de Java.

Mediante opciones de compilación se puede generar con *C-OOL* un modelo de tareas para que *ATOMS* aporte soporte run-time a la aplicación de simulación [Alfo98a]. Este soporte puede ser, por ejemplo ayuda automática sobre la interfaz, tutoring automático sobre determinadas acciones, soporte de macros, etc.

El compilador también genera documentación del modelo compilado (ver sección VIII.4) y las páginas *HTML* de los cursos (ver capítulo VIII).

El esquema de funcionamiento de *C-OOL* queda reflejado en la siguiente figura:

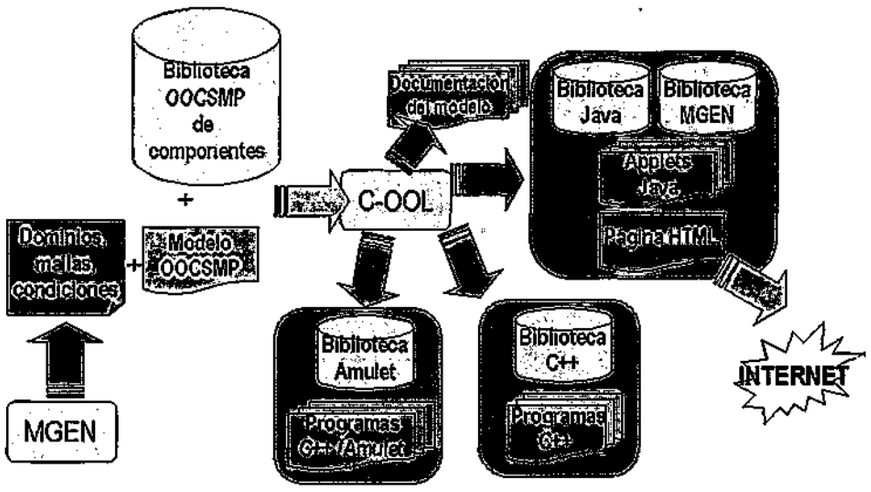


Figura IX.1: Esquema de funcionamiento de *C-OOL*

En el dibujo, el color claro indica programas y librerías que se han construido como parte del presente trabajo, a saber:

- *MGEN*.
- *C-OOL*.
- Las bibliotecas para Amulet, C++ y Java para los objetos de la interfaz, resolución numérica y generación de mallas.
- Bibliotecas para *MGEN*, de soporte de objetos gráficos.
- Una biblioteca de componentes *OOCSMP*.

Las cajas redondeadas son compiladores o generadores de código. Las flechas de color oscuro, significan una generación de código automática. Las cajas con forma de folios indican código generado automáticamente. Las flechas significan un proceso de compilación (el código Java generado por *C-OOL* se debe compilar con algún compilador de Java antes de ponerlo en la web).

El usuario ha de programar sus modelos *OOCSMP*, y puede utilizar código *OOCSMP* generado por *MGEN*, o bien de las bibliotecas. Al compilar el modelo con *C-OOL* debe elegir mediante opciones de compilación si quiere generar código C++/DOS, C++/Amulet o Java. Con otras opciones, se puede configurar la interfaz de usuario, generar páginas *HTML*, documentación, eliminar los semáforos del código distribuido, etc. La siguiente tabla muestra todas las opciones de compilación admitidas por *C-OOL*.

Opción	Descripción	Lenguaje objeto
`-jdk1.1.0`	Genera código para JDK 1.1.0 (opción por defecto)	Java
`-jdk1.1.1`	Genera código para JDK 1.1.1	Java
`-package=<nombre>`	Genera el nombre de package especificado, por defecto genera el package *csmp<nombre-fichero-principal>*	Java
`-MACHINE=<maquina>`	Activa la generación de código paralelo, y genera los programas que serán utilizados en la máquina con etiqueta `<maquina>`.	Java
`-NOSEMAPHORES`	Si estamos generando código distribuido, no coloca semáforos.	
`-noFrame`	En lugar de generar un programa Java, genera un Applet	Java
`-noLeyenda`	No genera la ventanas o paneles con las leyendas que indica el color en que se representa cada variable.	Java/Amulet
`-noScaleWindow`	No genera las ventanas o paneles que indican las escalas de las representaciones.	Java
`-noPrintWindow`	No genera la ventana o panel en el que se muestran las variables especificadas en la sentencia *PRINT*, pero se continuarán imprimiendo en la consola.	Java/Amulet
`-noButtons`	No genera los botones que abren las ventanas de cambio de parámetros de los distintos objetos de simulación y de las variables globales.	Java/C++
`-printOnly`	Genera una tabla con las variables especificadas en la instrucción *PRINT* en el espacio que normalmente se reserva para la representación gráfica.	Java
`-addObjects`	Genera botones que permiten añadir/borrar objetos a la simulación. Estos objetos han de ser de alguna de las clases que se han usado en el problema. También permiten añadir/borrar/intercambiar objetos de colecciones de objetos.	Java/Amulet
`-Width=<ancho>`	Ancho de la ventana principal de Java en pixels	Java
`-Height=<largo>`	Largo de la ventana principal de Java en pixels	Java
`-HTMLpage=<pagina>`	Genera una página *HTML* con las instrucciones *SODA*.	Java
`-genDoc`	Genera páginas *HTML* con la documentación del modelo.	Java/C++/Amulet
`-genConnPlots`	Permite la generación de gráficos icónicos (mandato *CONNECTIONPLOT* de *OOCSMP*).	Java
`-delay`	Genera un retardo, configurable desde la ventana de cambio de parámetros	Java
`-step`	Genera un botón para ejecutar paso a paso la simulación	Java
`-C++`	Genera código C++ sólo para la función de simulación.	C++
`-DOS`	Genera código C++ junto con una interfaz para *DOS*	C++
`-Amulet`	Genera código C++ junto con una interfaz que usa la librería Amulet v3.0.	Amulet
`-genTasks`	Genera un modelo de tareas sobre la interfaz generada con la opción `-Amulet`. Este modelo puede ser utilizado por *ATOMS* para aportar soporte run-time a la aplicación generada.	Amulet

Tabla IX.1: Opciones de compilación de *C-OOL*.

Las siguientes secciones describen cada una de estas opciones, agrupadas por el lenguaje objeto que se genera, así como el funcionamiento de la interfaz en cada caso.

IX.2 Generación de código C++/DOS

El compilador genera código C++ en caso de que se introduzcan las opciones *-DOS* o *-C++*. La opción *-DOS* genera una interfaz para *DOS*, las bibliotecas gráficas que proporcionamos son compatibles con el Turbo C para *DOS* de Borland [Schi90].

Tanto si se compila con la opción *-DOS*, como con la opción *-C++*, se generan los siguientes ficheros:

- Un programa C++ para el modelo principal, de nombre *<nombre_prog_OOCSMP>.cpp*.
- Para cada clase definida o incluida en el modelo, se genera un fichero de cabecera (extensión .hpp) y un fichero C++ (extensión .cpp) llamados *_<nom_clase_OOCSMP>.hpp* y *_<nom_clase_OOCSMP>.cpp*

Para obtener un ejecutable, se deben compilar los ficheros generados junto con los módulos:

- vector.cpp
- matrix.cpp
- oocsmp.lib

que son comunes para todas las simulaciones. Además los ficheros:

- menu.hpp
- barra.hpp
- vector.hpp
- matrix.hpp
- csmp.h

Deben estar accesibles en el *path* donde el compilador de C++ busque los ficheros que incluye.

IX.2.1 La interfaz para *DOS*

La interfaz para *DOS* se genera al compilar un programa *OOCSMP* con la opción *-DOS*. Los objetos gráficos que incluye esta interfaz son:

- Un botón por cada objeto de la simulación. Estos botones tienen como propósito abrir una ventana que permite cambiar los parámetros de dichos objetos, así como inhibir su dibujo en la gráfica.
- Un botón para cambiar los parámetros globales.
- Un botón para cambiar el paso elemental de tiempo, el intervalo de impresión y el intervalo de dibujo de las variables.
- Una barra de desplazamiento que permite cambiar el tiempo final de la simulación.
- Una zona en la que se dibuja como mucho un gráfico bidimensional (instrucción *PLOT*) animado. Es decir, un modelo *OOCSMP* que además tuviera otro tipo de gráficos, y se compilara a C++, no generaría más que el primer gráfico definido por la primera instrucción *PLOT*. Por tanto, los parámetros de posición de las instrucciones gráficas se ignoran.

Como primer parámetro del programa C++ generado, se puede indicar el nombre de un fichero. Esto provoca que el contenido de las variables que se especificaron en el comando *PRINT* se vuelquen en el fichero.

La siguiente figura (IX.2) muestra un ejemplo de una interfaz generada para *DOS* del problema de la gravitación con tres planetas (Marte, la Tierra y Venus).

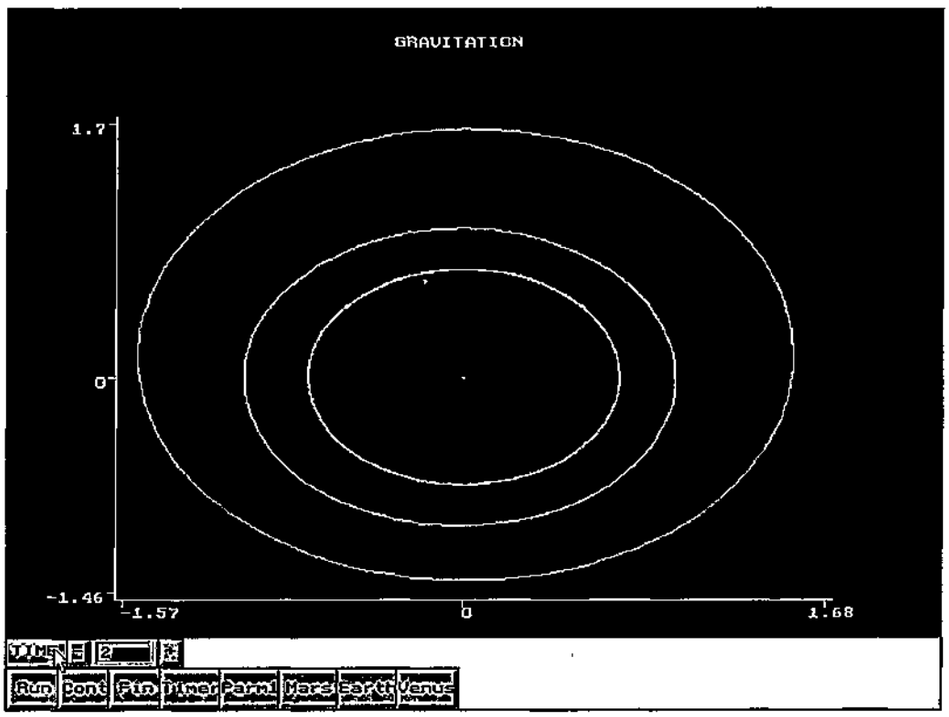


Figura IX.2: Interfaz *DOS*.

IX.3 Generación de código C++/Amulet

Cuando llamamos al compilador con la opción de compilación *-Amulet*, *C-OOL* genera una interfaz que usa la biblioteca Amulet [Myer97] [Open97]. Amulet (Automatic Manufacture of Usable and Learnable Editors and Toolkits) se ha desarrollado en la universidad de Carnegie Mellon. Las aplicaciones construidas con estas bibliotecas pueden portarse sin modificaciones a Unix, PC y Macintosh. Amulet proporciona herramientas que facilitan la construcción de interfaces gráficos de manipulación directa. La apariencia de los objetos gráficos creados con esta biblioteca es independiente de la plataforma. También proporciona un sistema de objetos prototipo-elemento basado en restricciones construido sobre la jerarquía de clases de C++. Un prototipo de un objeto Amulet es otro objeto, no una clase. Las propiedades de los objetos Amulet se almacenan en *slots*. Amulet comprueba dinámicamente los tipos, un *slot* Amulet puede contener cualquier tipo.

Esta interfaz consta de los mismos elementos que la interfaz *DOS* y además:

- Una ventana en la que se imprimen las variables especificadas en el comando *PRINT*. Esta ventana se puede inhibir con la opción *-noPrintWindow*.
- Una ventana de leyenda, en la que se indica el color con el que se representa cada variable. Esta ventana se puede inhibir mediante la opción *-noLeyenda*.
- Botones para añadir/borrar objetos de la simulación, estos botones aparecen cuando se compila con la opción *-addObjects*.
- Un menú desplegable, en el que se tiene acceso a las mismas acciones que desde los botones, pero, por ejemplo, los botones para cambiar el valor de los atributos de los objetos están agrupados en menús, ordenados por la clase de cada objeto.

Al igual que en el caso C++/*DOS*, la biblioteca de objetos gráficos que hemos construido para Amulet, sólo contiene el gráfico bidimensional (instrucción *PLOT*).

La siguiente figura muestra un ejemplo de una interfaz Amulet con las opciones por defecto, para el problema de la gravitación, esta vez con cuatro planetas.

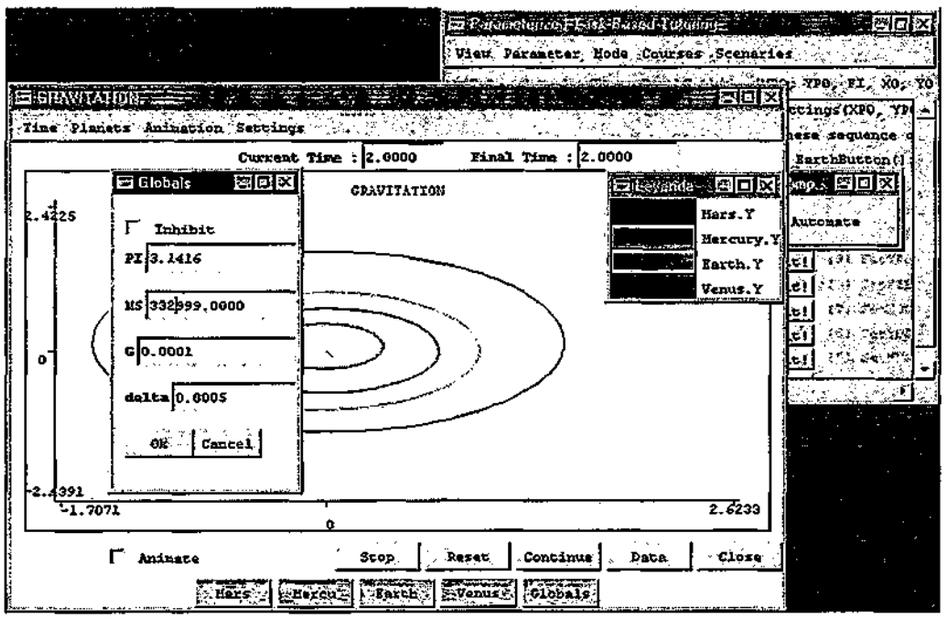


Figura IX.3: Una interfaz *Amulet* para el problema de la gravitación.

En el ejemplo anterior, además se ha usado la opción *-genTasks*, para que el compilador genere automáticamente un modelo de tareas [Alfo98a] que pueda ser usado por *ATOMS*, para que este ofrezca ayuda sobre las acciones que se pueden realizar en la interfaz. *ATOMS* [Garc98] [Fede99]

[Rodr97] presta esta ayuda mediante *HATS* (Help for ATOMS Task System), que es el subsistema de ayuda de *ATOMS*. *HATS* es capaz de guiar al usuario a través del aprendizaje de la interfaz, indicando qué pasos se han de dar para hacer algo o qué opciones tiene el sistema disponible en cada paso. En la figura anterior, IX.3, la interfaz de *HATS* aparece en segundo plano. Tanto *ATOMS* como *HATS* han sido construidos por Federico García y no son parte del presente trabajo.

También es posible que *C-OOL* genere archivos de escenarios. Los escenarios son conjuntos de tareas que *ATOMS* (su módulo de emulación de tareas) es capaz de ejecutar sin intervención del usuario, pero de forma idéntica a como las haría el usuario, es decir, un cursor simulado aparece en la interfaz manipulando sus elementos. De esta forma el usuario aprende mediante observación a manejar la interfaz y a establecer valores significativos a los parámetros de la simulación.

Los escenarios se definen por el programador de la simulación dentro del modelo *OOCSMP*, *C-OOL* los compila y genera los ficheros necesarios para que sean interpretados por *ATOMS*.

La arquitectura propuesta, para el trabajo conjunto de *C-OOL* y *ATOMS*, se muestra en la figura IX.4.

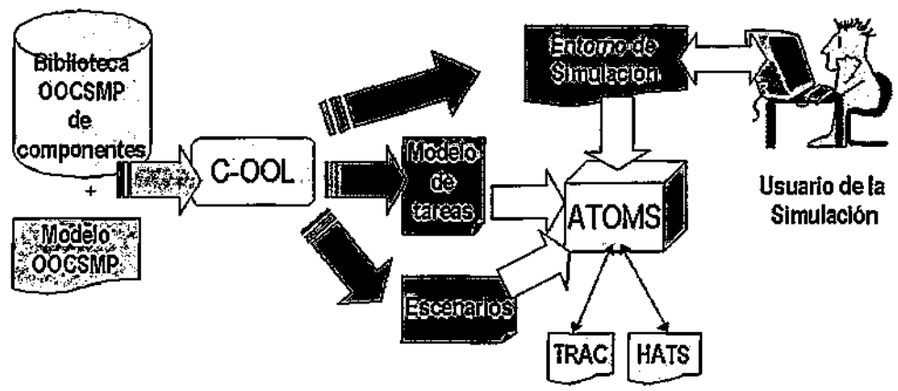


Figura IX.4: Arquitectura propuesta *ATOMS+OOCSMP*.

El siguiente listado, IX.1, muestra parte del modelo de tareas que generaría *C-OOL* automáticamente para el ejemplo de la gravitación anterior.

```
//Atomic task for global settings button
ATOMIC GlobalsButton
BUTTON "Globals"
DESCRIPTION Begin global properties editing by clicking on Globals button
END

//Atomic task for global settings menu option
ATOMIC GlobalsMenu
MENU "Globals"
DESCRIPTION Begin global properties editing by selecting Globals menu item
END

// Bring up the globals dialog
COMPOSED ChangeGlobals
DESCRIPTION Brings up the globals' dialog
END

// The dialog may be poped-up following these two methods:
RULE ChangeGlobals
SUBTASKS GlobalsButton AS Button GlobalsMenu AS Menu
SEQUENCING OR
END

//Accept global values
ATOMIC OkGlobalsButton
BUTTON "Ok"
```

```
DIALOG_TITLE == "Globals"
DESCRIPTION Accept new sum mass value
END

// Modify the text input box
ATOMIC SetSM
TEXT_BOX  == "SM"
DIALOG_TITLE == "Globals"
PARAMETER SM = TEXT
DESCRIPTION Sets SM property value
END

// High-level task of modify sun mass value
COMPOSED ChangeSMGlobalSetting
DESCRIPTION Allows the SM property global setting to be changed
END

RULE ChangeSMGlobalSetting
SUBTASKS ChangeGlobals AS Change SetSM AS SetValues OkGlobalsButton AS Accept
SEQUENCING SEQ
MULTIPLE SetValue
PARAMETER SM = SetValues.Params.SM
END
```

Listado IX.1. Parte del modelo generado para un conjunto de tareas.

Como se puede observar, por ejemplo, se puede cambiar la masa del Sol (variable *SM*), apretando el botón *'Globals'*, y mediante la opción equivalente del menú. Igualmente, la tarea de modificar sus parámetros necesita que se complete la secuencia de acciones señaladas:

- Seleccionar la opción de modificar los valores del Sol, independientemente del procedimiento utilizado.
- Introducir el valor deseado en el campo destinado a tal efecto, acción que puede realizarse una o más veces, para permitir la corrección de errores al teclear.
- Por último, confirmar la operación.

En cuanto a los parámetros, el único de esta tarea es el nuevo valor de la masa solar, que se obtiene a partir del objeto sobre el que el usuario ha interaccionado al teclear el nuevo valor y se transforma en el parámetro de la tarea de nivel superior sin sufrir modificación alguna.

Es posible preparar situaciones interesantes en la simulación, tales como aumentar la masa del Sol, añadir un nuevo planeta, etc. Para ello, se ha sobrecargado la instrucción "\" de *OOCSMP*. Tras esta instrucción, se ha de indicar el nombre del escenario, y a continuación, las acciones que se quieren preparar.

```
\ APOLLO
Planet Apollo ("Apolo",1957E-14, 0, 1.4849,-4.253,  2.915, 6.4  )
Planet InnerSys :=Mercury,Venus,Earth,Moon,Mars,Venus, Apollo, Jupiter
\ SUNMASS
MS := 400000
Mars.M := 0.15
```

Listado IX.2: Código *OOCSMP* correspondiente a la definición de un escenario.

En el listado anterior se han preparado dos escenarios, uno en el que se añade el cometa Apolo a la simulación del sistema, y otro en el que se aumenta la masa del Sol y del planeta Marte. El compilador de *OOCSMP* en este caso, generaría dos ficheros (*APOLLO* y *SUNMASS*) con las tareas que habría que realizar sobre la interfaz para llevar a cabo las acciones anteriores. Además, la ejecución de estos escenarios se realizaría de forma animada, de forma que el usuario puede ver qué interacciones hay que hacer sobre los elementos de la interfaz para alcanzar el objetivo.

■ IX.4 Generación de código Java

La opción por defecto de *C-OOL* es la generación de código Java. Para este caso, el compilador genera dos ficheros Java:

- *<nombre_programa_OOCSMP>.java* programa principal, una subclase de *Applet* que implementa los métodos *init()* y *main()*, de esta forma puede ejecutarse como programa o como *applet*, la simulación se ejecuta en una ventana Java.
- *frm_<nombre_programa_OOCSMP>.java* una subclase de *Frame*.

Además, por cada clase definida o incluida en el modelo mediante *INCLUDE*, se crea una clase Java.

Otra de las opciones (*-noFrame*) consiste en generar un *applet*, en este caso el resultado de la compilación del modelo es un solo archivo Java (el segundo de la lista anterior, que en este caso subclasificaría la clase *Applet* directamente). Aquí el *applet* se ejecuta directamente en un panel dentro de la página *HTML*. Esta es la opción que se ha usado para compilar los modelos de los cursos descritos en el capítulo X.

Además siempre que se incluya o defina algún objeto, se genera un archivo llamado *controlVars.java*, que es un archivo de interfaz con las variables y procedimientos definidos en el modelo principal *OOCSMP*.

Todos los archivos Java generados, por defecto se incluyen en un *package* Java con nombre: *csmp<nombre_programa_OOCSMP>*, si bien esto puede cambiarse mediante la opción *-package=* del compilador.

Si se compila con la opción de generar una página *HTML*, y no hay instrucciones *SODA*, esta contendrá un título y una referencia al applet generado, es decir, contendrá el tag *HTML*:

```
<applet
      code ="csmp<nombre_OOCSMP>.frm_<nombre_OOCSMP>.class"
      WIDTH=650
      HEIGHT=650>
</applet>
```

Sintaxis IX.1: Tag *HTML* que contiene al applet.

Los programas Java generados por *C-OOL* deben ser ubicados en principio (debido al nombre de paquete con que se generan), en un directorio llamado *csmp<nombre_programa_OOCSMP>* a partir del path que indique la variable de entorno *CLASSPATH*.

Las bibliotecas Java que se suministran se deben incluir también a partir de ese mismo path, como se indica en el siguiente esquema, suponiendo que se ha compilado un modelo *OOCSMP* llamado ***grav.csm*** y que ***CLASSPATH*** tiene el valor ***c:\classes***.

```
c:\classes\csmpgrav>
                grav.java
                frm_grav.java
                controlVars.java
c:\classes\csmp>
c:\classes\csmp\plot>      <LIBRERÍA GRAFICA>
c:\classes\csmp\pde>       <LIBRERÍA PARA PDEs>
c:\classes\csmp\math>      <LIBRERÍA MATEMATICA>
c:\classes\csmp\util>      <LIBRERÍA DE MANEJO DE STRINGS>
c:\classes\mgen>           <LIBRERÍA DE GENERACION DINAMICA DE MALLAS>
```

Esquema IX.1: Directorios en los que hay que colocar los programas generados y las librerías.

La tarea de generar código [Aceb97] de los modelos *OOCSMP* es bastante distinta, dependiendo del lenguaje objeto (C++ o Java). La mayor diferencia, que ya se ha visto, está en la interfaz gráfica que se genera. Algunos de los objetos gráficos de Java no son posibles en el caso de *C++*. El siguiente listado muestra un esquema del código que se genera en el caso de Java.

```
package csmp<NAME>
import java.awt.*;
...
//import objects from our Java library
import csmp.plot.PlotData;
...
public class frm_<NAME> extends (Frame|Applet) implements Runnable [ ,...]
// other interfaces, depending on the graphical outputs selected...
{
// Declare arrays of simulated pointers to the variables beeing integrated,
// plot and printed ...
...
// Declare the model Data
...
// Declare the graphical objects
...
public void run() // launchs a thread for the calculus
{ ... }
public void stop()// Stops the thread
{ ... }
void <NAME>_s2()  // implements the calculus to be done in the simulation loop
{ ... }
void initAllArrays()// initializes the arrays of  simulated pointers...
{ ... }
public void frm_<NAME>()
// constructor,adds graphical objects and initializes data
{ ... }
public void <NAME>_sim( ... ) // The simulation Loop
{ // Initialize the selected graphical representations
...
for (;;){
      <NAME>_s2();
      // Print the selected variables
      ...
      // Plot some variables (varies depending on the chosen graphical output)
      ...
      // Perform integration, depending on the selected integration method...
      ...
}
}
public boolean handleEvent(Event e) // Handles user actions
{...}
private void updateArrays()
// Updates the array of pointers to the variables beeing integrated
{...}
private void updatePlots() // Updates the array of variables to be plotted
{...}
private void updatePrints() // Updates the array of variables to be printed
{...}
// Some other functions depending on the graphical outputs selected
}
```

Listado IX.3 : Un esquema del código Java generado por *C-OOL*.

Otra diferencia importante es que en Java no existen punteros, y que no es posible el paso de parámetros por referencia. Esto se ha solucionado de la siguiente forma:

- La no existencia de punteros se ha solucionado simulándolos. El contenido de los punteros se actualiza en los momentos oportunos. Esto es necesario, por ejemplo, debido a que se tienen vectores que 'apuntan' a las variables que se imprimen, o que se dibujan. En C++ no hay problema, debido a que los elementos de estos vectores son punteros. En Java los elementos de

los vectores son de tipo numérico, se ha de modificar su valor justo antes de proceder a la representación de las variables.

- En cuanto al paso de variables por referencia en Java, se ha tenido que crear dos clases *'wrapper'* que encapsulan datos de tipo *entero* y *doble*. En lugar de pasar directamente una variable de estos tipos, se pasa un objeto de las clases construidas, que sí pueden cambiar su valor, mediante los métodos adecuados.

Otras diferencias son, por ejemplo, que los atributos de una clase C++ no se pueden inicializar en su declaración, mientras que en Java sí es posible, incluso referenciando atributos que se declaran posteriormente.

IX.4.1 La interfaz para Java

En la ejecución de la simulación, se crea un hilo para la rutina de simulación y la de representación gráfica, y otro hilo para dar respuesta a las acciones del usuario en la interfaz. Además, como ya se dijo en el apartado VI, algunas de las salidas gráficas crean su propio hilo. En la interfaz Java, se generan los mismos componentes que para la interfaz *DOS* anterior y además:

- Es posible generar todas las salidas gráficas expuestas en el apartado VI y combinarlas en un mismo problema.
- Se genera un botón para interrumpir la simulación -esto permite cambiar algún parámetro de simulación, y continuarla después- y otro para reiniciarla (las variables toman otra vez sus valores iniciales). También se puede generar un botón para ejecutar la simulación paso a paso (opción –*step*)

Es posible cambiar el valor de variables durante la simulación, pulsando el botón '*globals*' (para cambiar variables globales del modelo), o bien en los botones con los nombres de los objetos de la simulación. Se pueden cambiar variables de tipos numérico, vector numérico y matriz numérica. También se pueden modificar los objetos que son parte de otro objeto. Si se compila con la opción de añadir/borrar objetos, también es posible mover los objetos entre las distintas colecciones (ver sección IX.5)Una ejemplo de ventana para la modificación de variables es la siguiente :

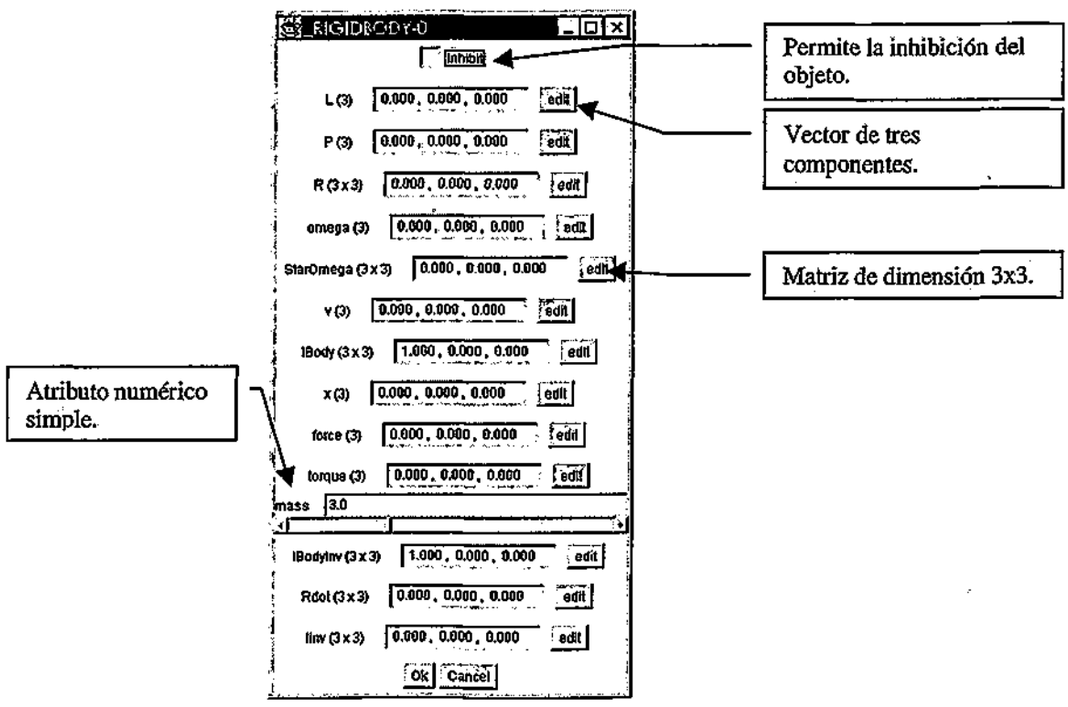


Figura IX.5: Una ventana para el cambio de variables.

Para el cambio de variables de tipo vector o matriz, es necesario pusar el botón `"edit"` asociado. En el recuadro de texto asociado a una matriz, se muestran los componentes de la primera fila. La figura IX.6 muestra una ventana para el cambio de los valores de una matriz. La ventana para el cambio de un vector es similar, pero sin los botones para moverse entre filas. El cambio de valores de variables simples se puede hacer desde la ventana de la figura IX.5, mediante la barra de desplazamiento.

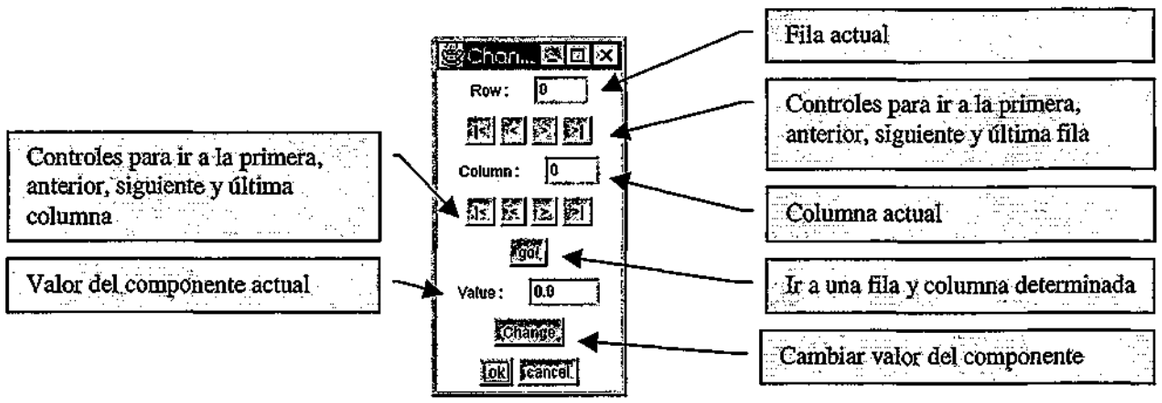


Figura IX.6: Ventana para modificar matrices.

También se puede cambiar el icono asociado a un objeto, desde la ventana de cambio de parámetros. La siguiente figura muestra una situación en la que se está cambiando el icono asociado a un objeto en un modelo ecológico. El campo "*Icon File*" es un fichero que contiene un índice con los iconos que podemos elegir (una línea por cada *URL*). Este fichero se puede cambiar mediante el botón "*change*".

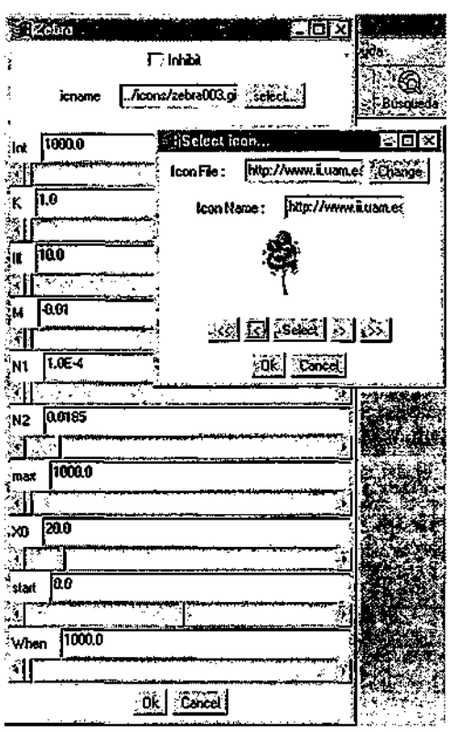


Figura IX.7: Ventana para modificar el icono asociado a un objeto.

IX.5 Manipulación de los objetos durante la simulación

Las interfaces Java y Amulet permiten añadir nuevos objetos, borrar objetos de la simulación o cambiar de colección los objetos existentes [Alfo99b]. Esta característica, por ejemplo, nos permitiría añadir o borrar planetas de las simulaciones de los n-cuerpos, añadir una nueva especie a una simulación ecológica, añadir una nueva pieza al problema del calentamiento de piezas (sección V.6.1), etc. Por el momento esto sólo es posible si la simulación no es distribuida.

Los botones (y las acciones asociadas) para la manipulación de objetos durante la ejecución de la simulación se generan mediante la opción de compilación *-addObjects*. Se generan cuatro botones, etiquetados como *newObj*, *delObj*, *addArray* y *delArray*, y sirven para añadir un objeto, borrar un objeto, añadir un objeto a una colección y borrar un objeto de una colección.

El siguiente ejemplo es una simulación del sistema solar interior, que se ha compilado de forma que se puedan añadir o borrar planetas. El modelo de esta simulación se muestra en el listado IX.4, y el resultado de ejecutar dicho modelo sin añadir ningún objeto se muestra en la figura IX.8.

```
* Universal data
DATA G:=0.00011869, PI:=3.141592653589793
* Sun data
DATA MS:=332999
INCLUDE "Planet.csm"
* Actual planets
Planet Mercury("Mercu",0.055271,-0.3871, 0,      2.078, -9.892, 7.004)
Planet Venus   ("Venus",0.81476,  0.7233, 0,      0.051,  7.39,  3.394)
Planet Earth   ("Earth",1,        0,      1,     -6.2899, 0.107, 0    )
Planet Moon    ("Moon", 0.01235,  0,      0.9975,-6.0783, 0.107, 0    )
Planet Mars    ("Mars", 0.10734,  1.5233, 0,      0.476,  5.071, 1.85 )
Planet Apollo  ("Apolo",1957E-14, 0,      1.4849,-4.253,  2.915, 6.4  )
Planet Jupiter("Jupit",317.94,    0,     -5.2028, 2.754,  0.131, 1.308)
Planet InnerSystem := Mercury, Venus, Earth, Moon, Mars, Apollo, Jupiter
DYNAMIC
  InnerSystem.STEP()
  InnerSystem.ACTION(InnerSystem)
* Time intervals and other data
TIMER delta:=.0005, FINTIM:=2, PRdelta:=.1, PLdelta:=.01
METHOD ADAMS
```

Listado IX.4: Modelo del sistema solar interior

Cuando la simulación está ejecutándose, el usuario puede pararla (botón *stop*), y añadir un nuevo planeta (botón *newObj*). Esto hace que surja un cuadro de diálogo (en nuestro ejemplo surge uno como el de la figura IX.9a, ya que sólo usamos la clase *Planet*) preguntando de qué clase ha de ser el objeto.

Una vez que elegimos que sea de tipo *Planet*, hemos de completar un cuadro de diálogo en el que hay que indicar los valores iniciales de los parámetros del objeto. El cuadro de diálogo de nuestro ejemplo sería como el de la figura IX.9b. En esta figura hemos rellenado el cuadro de diálogo con los datos de Saturno. Una vez rellenado este cuadro de diálogo, tenemos que seleccionar la colección (si hubiera alguna en el modelo) donde queremos añadir el objeto. En nuestro caso lo vamos a añadir a la única colección: *InnerSystem* (figura IX.9c).

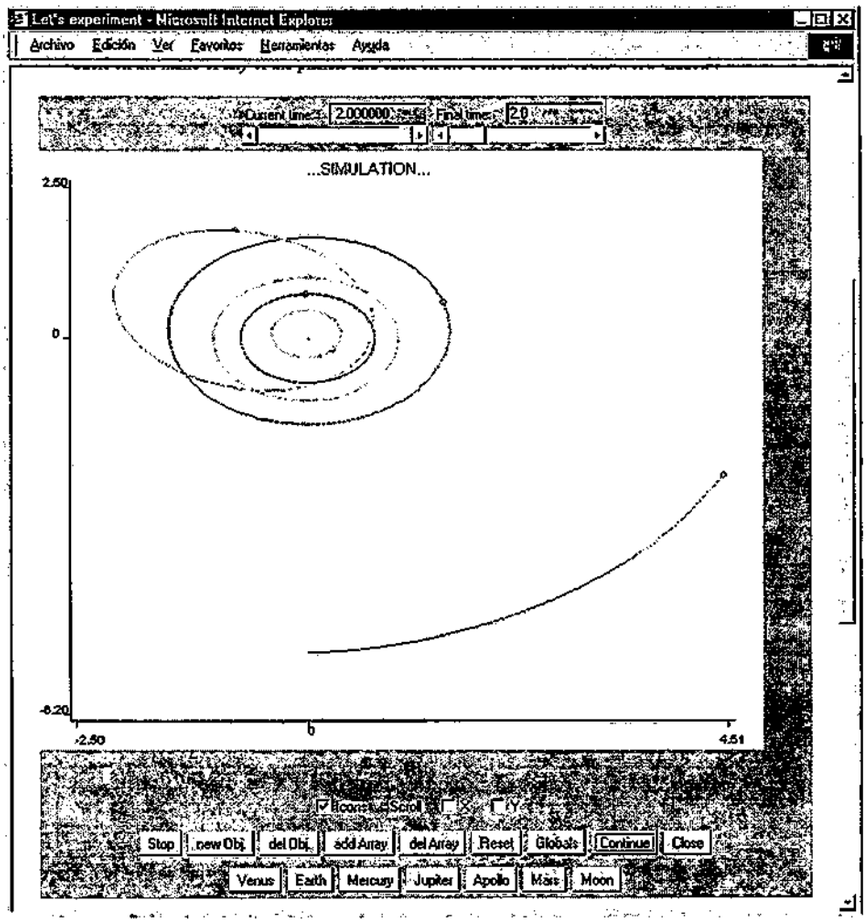
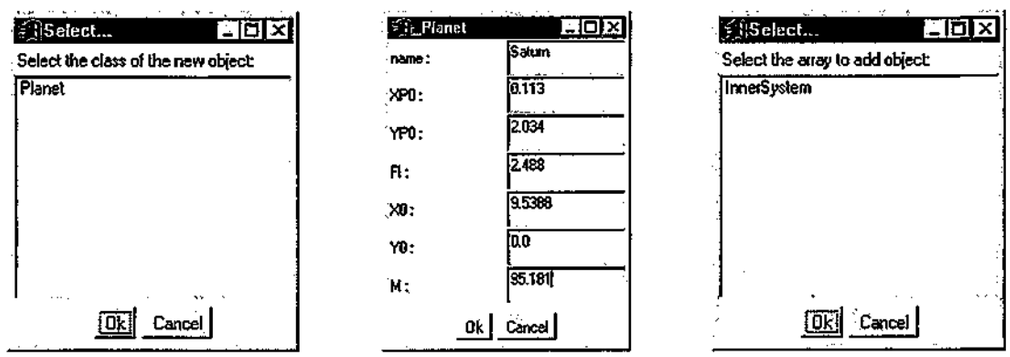


Figura IX.8: Simulación del sistema solar interior con los botones para crear nuevos objetos.



(a) (b) (c)

Figura IX.9a,b,c: (a) Elección de la clase del nuevo objeto, (b) Introducción de los datos de Saturno, (c) añadiendo Saturno a la colección de objetos

Una vez hecho esto, el planeta se ha añadido a la simulación, como se puede observar en la figura IX.9, se ha creado un botón etiquetado como "*Saturn*", y se puede observar su órbita en el gráfico (la más externa).

De forma similar se borrarían objetos de la simulación (su botón para el cambio de propiedades también se elimina de la interfaz, de la ventana de leyenda, etc.).

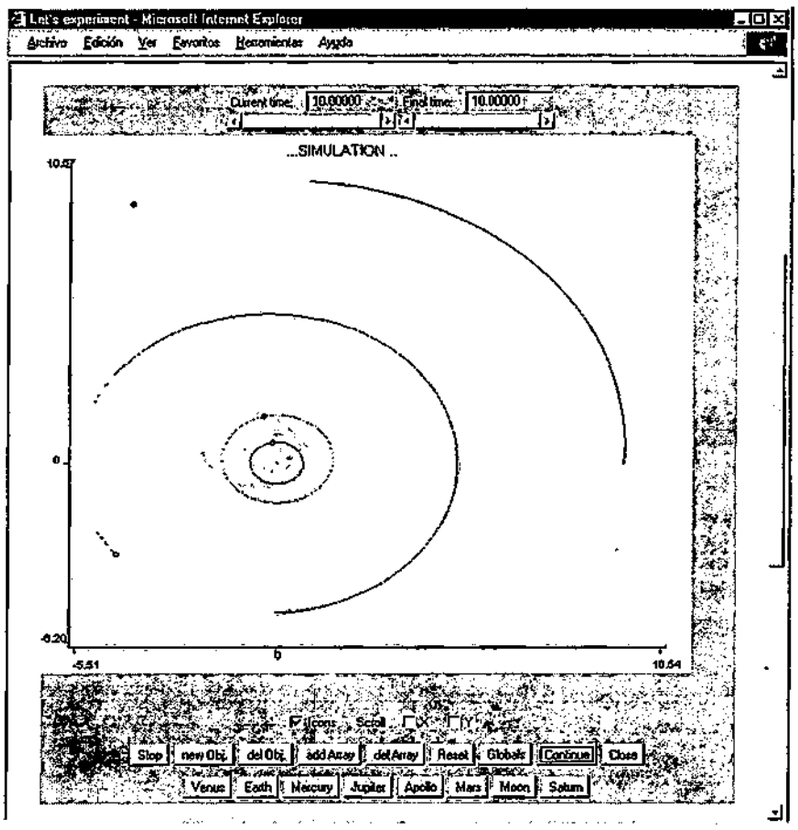


Figura IX.10: Simulación con Saturno creado dinámicamente

La posibilidad de añadir o borrar objetos de la simulación permite una mayor interacción del alumno con el modelo, y un mayor grado de experimentación, ya que no sólo es posible cambiar parámetros durante la simulación. En la página anterior, que se ha incluido en el curso sobre gravitación, también se proponen otros experimentos mediante el cambio de parámetros, tales como cambiar el valor de la constante de gravitación universal, cambiar la masa del Sol, cambiar la velocidad de la Tierra, etc. Con la posibilidad de añadir nuevos objetos, el conjunto de posible experimentos aumenta, siendo posible responder a preguntas del tipo: "¿Qué pasaría si añado un planeta cerca de la órbita de algún otro?", "¿Qué pasaría si añado un cuerpo de masa similar a la del Sol?".

Es necesaria cierta precaución al realizar estos experimentos, porque es posible que algunos de los fenómenos que ocurran sean debidos a que el experimento que se está efectuando haga inestable el método de integración.

IX.6 Compilación de una simulación distribuida

Mediante la opción de compilación '*-MACHINE=*' y la inclusión de las opciones *[MACHINE=<etiq-maquina>]* dentro de los modelos *OOCSMP*, es posible generar código distribuido. Esta opción sólo está disponible cuando se genera código en Java, ya que la distribución se efectúa utilizando las librerías *rmi* de Java.

El modelo distribuido se ha de compilar una vez por cada máquina que vaya a intervenir en la simulación, el identificador de la máquina para la que se compila el código se especifica en la opción *-MACHINE=* . Se crean los siguientes ficheros:

- *<nombre-maquina><nombre-modelo>.java,* con la misma funcionalidad que en el caso serie. Es el fichero principal, que se ejecuta.
- *frm_<nombre-maquina><nombre-modelo>.java,* con la misma funcionalidad que en el caso serie.
- *controlVars.java,* con la misma funcionalidad que en el caso serie, y además declara la interfaz con los métodos remotos. Este archivo es el mismo para todas las máquina.
- *<nombre-maquina><nombre-modelo>_Dist.java,* este es el fichero con la mayor parte del código.
- Los archivos correspondientes a cada clase definida por el usuario en el modelo. Son los mismos para todas las máquinas.

Para la ejecución de una simulación distribuida, se han de seguir los mismos pasos que para la ejecución de una simulación serie, pero además, se debe:

- Iniciar el registro *rmi* (mediante el programa *rmiRegistry*) en un directorio en el que tenga acceso a las clases generadas por *C-OOL*.
- Compilar los *stubs*, mediante el comando:

```
rmic <nombre-maquina><nombre-modelo>_Dist
```

Por ejemplo, para compilar el modelo de la sección VII.3.2, suponiendo que lo hayamos guardado en un fichero llamado *Pieces.csm*, se debería teclear:

```
C-OOL -MACHINE= m1 -NOSEMAPHORES Pieces
C-OOL -MACHINE= m2 -NOSEMAPHORES Pieces
C-OOL -MACHINE= m3 -NOSEMAPHORES Pieces
```

Ejemplo IX.1: Compilación del ejemplo de la sección VII.3.2

de esta forma obtendremos los programas Java para cada una de las máquinas. Con la opción *-NOSEMAPHORES* estamos inhibiendo la sincronización entre máquinas, ya que, somo se vio en el apartado VII.3.1, no hace falta en este caso porque no hay dependencias entre objetos y se quiere evitar que las máquinas esperen en el punto que sincroniza el tiempo de simulación en las distintas máquinas.

X. Los cursos generados para Internet

En este capítulo se presentan los cursos que hemos generado automáticamente para Internet, cada uno se explica en una sección separada, y son :

- Un curso sobre gravitación.
- Un curso sobre ecología.
- Un curso sobre electrónica básica.
- Un curso sobre *PDEs*.

Antes, en la sección X.1, se presenta un procedimiento que hemos desarrollado para la generación de cursos para Internet, y que hemos aplicado a estos cuatro cursos.

X.1 Introducción

A continuación se describen varios de los cursos que hemos generado con *OOCSMP* y *C-OOL*. Todos ellos se han construido siguiendo el procedimiento de la figura X.1.

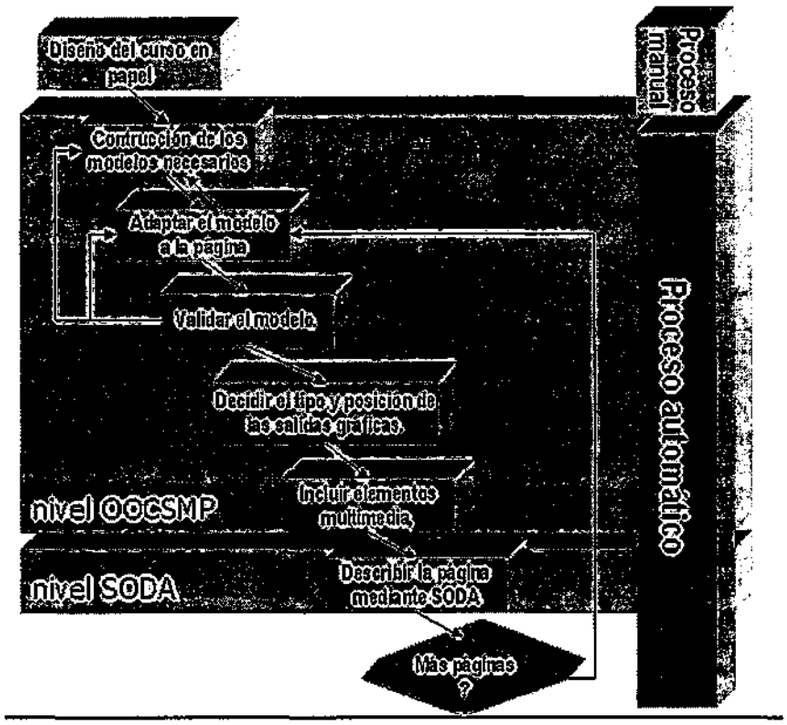


Figura X.1: Procedimiento para la construcción de los cursos.

1. El primer paso es diseñar el curso en papel. Dependiendo del curso, puede ser que usemos uno o más modelos, con alguna modificación en alguna página del curso. Es decir, en este paso se ha de decidir cuántas páginas va a haber, y qué modelo vamos a usar en cada página.
2. Ahora hay que construir los modelos, usando el lenguaje *OOCSMP*. Por ejemplo, en el curso de gravitación, en este paso hemos creado una clase *OOCSMP* (*Planet*) que encapsula el comportamiento de un planeta.
3. El siguiente paso es realizar las modificaciones necesarias a cada modelo para ajustarlo a cada página del curso. Por ejemplo, en el curso de la gravitación, hemos creado modelos del sistema solar interior, exterior, el sistema tierra-luna, etc. que utilizan la clase *Planet*. En cada caso hemos creado un número distinto de planetas, con distintos valores de sus atributos.
4. Para cada página, se valida el modelo. *C-OOL* genera una interfaz fácil de usar que nos permite probar rápidamente los modelos. Para validar los modelos, normalmente se elige como salida un listado de las variables que nos interesan.
5. Hay que escoger las salidas gráficas y su posición en la interfaz, teniendo en cuenta que se pueden incluir varias de ellas en un mismo problema.
6. Se incluyen los elementos multimedia. Para ello se debe seguir el procedimiento presentado en la sección VI.3 (figura VI.9).
7. Finalmente, se añaden las instrucciones *SODA* para la generación de las páginas. Normalmente estas instrucciones se incluyen en un archivo aparte, si bien pueden combinarse en un mismo archivo. En esta etapa hay que prever si se van a usar índices o trozos de descripción de páginas comunes. Si es así, es conveniente colocarlas en un archivo separado, para su posterior reutilización.
8. Se repite el proceso 3-8 para cada página del curso.

En las siguientes secciones se explican detalladamente cada uno de los siete pasos, mediante los ejemplos de los cursos que hemos generado con nuestra herramienta.

X.2 Un curso sobre la ley de la gravitación de Newton

Este curso [Alfo97], [Alfo98b], [Alfo99d] se puede encontrar en la dirección:

http://www.ii.uam.es/~epulido/newton/grav.htm

Paso 1: Diseño del curso en papel

El curso constará de siete páginas. Seis de ellas están basadas en el modelo del problema de los n-cuerpos. Este modelo se aplica en las páginas a distintas situaciones que dan una visión de los posibles usos de la ley de la gravitación de Newton. El modelo define una única clase (`Planet`) que se puede usar indistintamente para representar los planetas del sistema solar, o bien satélites naturales o artificiales. La otra página es una simulación de un sistema para el control de una serie de instrumentos instalados sobre un satélite. Las páginas que componen el curso son:

- Una descripción de la mecánica de Newton con diversas soluciones al problema de los dos cuerpos (una caída libre que sigue distintas órbitas: un círculo, una elipse, una parábola, una hipérbola y una caída libre en línea recta).
- Un modelo del sistema solar, como ejemplo práctico del problema de los n-cuerpos. Por conveniencia, el sistema solar se muestra en dos partes separadas: el sistema interior (de Mercurio a Júpiter) y el exterior (de Júpiter a Plutón), con diferentes escalas de tiempo.
- Un modelo del sistema Sol-Tierra-Luna. Las escalas de tiempo y del gráfico se tendrán que ajustar para poder distinguir las dos órbitas.
- El descubrimiento de Neptuno por John Couch Adams y Urbane Jean-Joseph Le Verrier. Se indica cómo este descubrimiento transformó un fallo aparente de la mecánica newtoniana en un gran éxito. El descubrimiento se ilustra con una simulación doble de la órbita de Urano, en presencia y ausencia de Neptuno.
- También se ha modelado un satélite geoestacionario que mantiene constante su distancia a la Tierra, usando el mismo modelo. Sólo se ha cambiado los valores de las constantes y de los atributos de los objetos (miembros de la clase `Planet`). El efecto de la Luna en la órbita del satélite se ilustra con una simulación doble, en presencia y ausencia de la Luna. Para ejecutar este segundo paso, sólo tenemos que cambiar la masa de la Luna a cero.
- Una página en la que se simula un sistema de control para el movimiento de una serie de instrumentos montados sobre un satélite.
- En esta página, vamos a dejar al estudiante que experimente libremente con la simulación del sistema solar. Para ello se le permite cambiar los valores de los parámetros de los planetas, e incluso las constantes universales (la masa del Sol o la constante de gravitación), así como añadir o borrar planetas a la simulación, para que experimente y responda a cuestiones del tipo *"qué pasaría si...?"*.

Paso 2: Construir los modelos necesarios

La mayor parte de este curso está basado en un modelo único de simulación del problema de los n-cuerpos, que es aplicado en diversas páginas a varias situaciones que dan una visión de los posibles usos de la ley de gravitación de Newton.

Es de hacer notar que si el número de cuerpos fuese muy pequeño, el modelo podría programarse fácilmente en cualquier lenguaje de propósito general o de simulación continua. Pero, cuando el número de objetos aumenta, el número de ecuaciones podría llegar a ser inmanejable. En estos casos, *OOCSMP* es muy útil, ya que simplifica notablemente la construcción del modelo.

En el listado VIII.6 se mostró la definición *OOCSMP* de la clase `Planet`, que se usa en todas las páginas del curso. La clase `Planet` implementa las ecuaciones de la mécanica Newtoniana en la sección `DYNAMIC` y en el método `ACTION`. En este modelo, se suponen que el Sol se coloca en el origen de coordenadas, y por tanto no tiene movimiento. Se ha incluido una sentencia `FINISH` que hace que se detenga la simulación cuando el planeta está demasiado cerca del Sol. Esto nos es útil, por ejemplo, para modelar la situación de la caida libre de la primera página. La clase además incluye la sentencia `PLOT`, que hace que se representen gráficamente todos los objetos de esta clase.

El segundo modelo que vamos a usar es el que nos va a servir para la simulación de un sistema de control para el movimiento de una serie de aparatos montados sobre un satélite. El modelo ha sido adaptado del que se puede encontrar en [Yoha69]. El código fuente *OOCSMP* se puede acceder desde la página del curso. Para este problema se van a preparar varias salidas gráficas (un diagrama de fases de la posición respecto a velocidad, un gráfico de la salida del amplificador, etc). El cambio de una salida a otra será mediante el diseño de simulaciones alternativas (instrucción '\'). La simulación será la misma, pero variará la salida gráfica.

❑ Paso 3: Adaptar el modelo a cada página

Si bien la clase `Planet` es la misma para todas las páginas del curso, se requieren variaciones en el número de cuerpos que se define en cada página, ya que:

- Para la página 1, se requieren dos cuerpos, y hay que adaptar los parámetros para conseguir cada una de las órbitas. El modelo de esta página puede encontrarse en el listado IV.3 (sección IV.9.3).
- Para la página 2, se requieren 7 (Mercurio, Venus, la Tierra, Marte, Júpiter, la Luna y Apolo) para el sistema solar interior y 5 (Júpiter, Saturno, Urano, Neptuno y Plutón) para el sistema exterior.
- Para la página 3 se necesitan dos cuerpos (la Tierra y la Luna),
- El modelo de la página 4 es el del sistema solar exterior, con y sin Neptuno.
- El modelo de la página 5 está compuesto de tres cuerpos (la Tierra, la Luna, y el satélite geoestacionario). Además, se ha de cambiar el valor de las constantes *MS* y *G*, porque la situación de los planetas cambia, el que está inmóvil es la Tierra (que hace de *'Sol'*), a su alrededor giran la Luna y el satélite.
- La última página muestra una simulación del sistema interior, en la que dejamos que el usuario experimente (vease el apartado IX.5).

Todos los modelos son bastante parecidos, lo único que varía es el número y las características de los planetas. Los modelos tienen un aspecto similar al de los listados VIII.5 y IX.4.

El modelo de la página del descubrimiento de Neptuno es algo distinta, ya que en el mismo modelo vamos a simular dos sistemas planetarios, el primero que contiene Neptuno, y el segundo sin él. Esto simula cómo se descubrió realmente la existencia de Neptuno, ya que sin Neptuno se produce una diferencia en la órbita de Urano que se hace notar a los 30 años (en la simulación, cuando *TIME* vale 30). Aquí se observa otra de las ventajas de la simulación, podemos acelerar el tiempo, y además podemos hacer suposiciones que nos van a ayudar a corroborar una teoría. El listado X.1 nos muestra el programa *OOCSMP* que hemos usado para este modelo.

```
DATA G:=0.00011869, PI:=3.141592653589793, MS:=332999
INCLUDE "Planet.csm"
* Actual planets                                              *
Planet Mercury("Mercu",0.055271,-0.3871, 0,      2.078, -9.892, 7.004)
Planet Venus  ("Venus",0.81476,  0.7233, 0,      0.051,  7.39,  3.394)
Planet Earth  ("Earth",1,        0,      1,     -6.2899, 0.107, 0    )
Planet Moon   ("Moon", 0.01235,  0,      0.9975,-6.0783, 0.107, 0    )
Planet Mars   ("Mars", 0.10734,  1.5233, 0,      0.476,  5.071, 1.85 )
Planet Apollo ("Apolo",1957E-14, 0,      1.4849,-4.253,  2.915, 6.4  )
Planet Jupiter("Jupit",317.94,   0,     -5.2028, 2.754,  0.131, 1.308)
Planet Saturn ("Satur", 95.181,  9.5388, 0,      0.113,  2.034, 2.488)
Planet Uranus ("Urano", 14.535,  0,     19.1914,-1.431,  0.067, 0.774)
Planet Neptune("Neptu", 17.135,-30.0611, 0,      0.0117,-1.147, 1.774)
Planet Pluto  ("Pluto",0.0021586,0,    -39.5294, 0.971 , 0.249,17.148)

Planet Mercury1("Mercu",0.055271,-0.3871, 0,      2.078, -9.892, 7.004)
Planet Venus1  ("Venus",0.81476,  0.7233, 0,      0.051,  7.39,  3.394)
Planet Earth1  ("Earth",1,        0,      1,     -6.2899, 0.107, 0    )
Planet Moon1   ("Moon", 0.01235,  0,      0.9975,-6.0783, 0.107, 0    )
Planet Mars1   ("Mars", 0.10734,  1.5233, 0,      0.476,  5.071, 1.85 )
Planet Apollo1 ("Apolo",1957E-14, 0,      1.4849,-4.253,  2.915, 6.4  )
Planet Jupiter1("Jupit",317.94,   0,     -5.2028, 2.754,  0.131, 1.308)
Planet Saturn1 ("Satur", 95.181,  9.5388, 0,      0.113,  2.034, 2.488)
Planet Uranus1 ("Urano", 14.535,  0,     19.1914,-1.431,  0.067, 0.774)
Planet Neptune1("Neptu", 0,-30.0611, 0,      0.0117,-1.147, 1.774)
Planet Pluto1  ("Pluto",0.0021586,0,    -39.5294, 0.971 , 0.249,17.148)

Planet System := Mercury, Venus, Earth, Moon, Mars, Apollo, Jupiter, Saturn, Uranus,
Neptune, Pluto
```

```
Planet System1 := Mercury1, Venus1, Earth1, Moon1, Mars1, Apollo1, Jupiter1, Saturn1,
Uranus1, Neptune1, Pluto1
DYNAMIC
 System.STEP()
 System.ACTION(System)
 System1.STEP()
 System1.ACTION(System1)
TIMER delta:=.0005, FINTIM:=60
METHOD ADAMS
```

Listado X.1: Modelo para simular el descubrimiento de Neptuno

Como puede observarse, en realidad ambos sistemas planetarios contienen al planeta Neptuno, pero uno de ellos tiene su masa igual a cero.

❑ Paso 4: Validar el modelo

En la validación de los modelos, hemos tenido que ir ajustando los pasos elementales de tiempo, para conseguir una precisión adecuada en las órbitas. Para ello, normalmente nos hemos servido de un gráfico bidimensional y un listado de las órbitas. Hemos tenido que comprobar, por ejemplo, que la posición de la Tierra cada año (e.d. cuando *TIME=0, 1, 2*, etc.) es la misma, y lo mismo para cada planeta, cada uno en función de su "año" respectivo.

Para el modelo del control del satélite, hemos comprobado los resultados con los de [Yoha69].

❑ Paso 5: Decidir el tipo y posición de las salidas gráficas

En casi todos los modelos vamos a usar el gráfico animado en dos dimensiones, con algunas excepciones:

- En la página uno, vamos a usar la instrucción '\' para incluir las simulaciones de los distintos tipos de órbita (ver el listado IV.3, sección IV.9.3).

- En la página 4, que es la que inluye la simulación del descubrimiento de Neptuno, mostramos dos tablas con datos de la órbita de Urano con y sin Neptuno. Se nos plantea un problema, y es que por defecto, todos los objetos de la clase *Planet* se incluyen en un gráfico bidimensional. Para evitar esto, se ha creado otra clase exactamente igual (llamada *Planet1*), sólo que sin la instrucción *PLOT*.

- En la página del satélite geoestacionario (página 5), también vamos a usar la instrucción '\' para mostrar la simulación con y sin la Luna. Se nos plantea un problema similar al de la página anterior, y es que aunque pongamos la masa de la Luna igual a cero en la simulación alternativa, no queremos que salga en el gráfico. Para evitar esto también vamos a usar la clase *Planet1*. De esta forma, el listado *OOCSMP* del modelo para esta página quedaría como sigue:

```
TITLE WITH MOON
DATA G:=4.979E-16, PI:=3.141592653589793, MS:=5.979E21
INCLUDE "Planet1.csm"
Planet1 Geost ("Geost",1,0,42.24637,-265.462,0,0)
Planet1 Moon  ("Moon",7.384E19,0,392.1,-86.65,4.4,0)
Planet1 SGM := Geost, Moon
DYNAMIC
  SGM.STEP()
  SGM.ACTION(SGM)
TIMER delta:=.0005, FINTIM:=2, PRdelta:=.1, PLdelta:=.01
PRINT [E],Geost.X
PLOT  [C],47,394,-47,-186,Geost.X, Geost.Y
PLOT  [C],47,394,-47,-186,Moon.X, Moon.Y
METHOD ADAMS
\
TITLE NO MOON
Moon.M := 0.0
PLOT [C], Geost.X, Geost.Y
```

Listado X.2: Modelo para la simulación del satélite geoestacionario.

Como se puede observar, en la simulación principal se han de incluir explícitamente ambos cuerpos en el gráfico, y en la simulación alternativa, sólo se dice que se pinte el satélite geoestacionario. Además, en ambas simulaciones vamos a generar un listado de la órbita del satélite geoestacionario.

En todos los modelos hemos de ajustar la escala del gráfico bidimensional, esto es especialmente importante por ejemplo en el modelo del sistema tierra-luna (ver figura X.2).

- En la página de la simulación del control del satélite, también vamos a usar la instrucción '\' para cambiar entre varias salidas gráficas.

❑ Paso 6: Incluir los elementos multimedia

Sólo vamos a incluir un panel de texto en la página del descubrimieno de Neptuno. Este panel de texto aparecerá cuando haya terminado la simulación (se simulan 60 años) y se colocará debajo de los dos listados de las órbitas de Urano. El código *OOCSMP* que es necesario incluir en el modelo X.2 se muestra a continuación (junto con el código necesario para mostrar los listados de las órbitas).

```
...
PRINT  [C], Uranus.X,Uranus.Y
PRINT  [E], Uranus1.X,Uranus1.Y
TEXTPANEL [S, SE], START( TIME >= FINTIM ), "explain.txt"
```

Listado X.3: Instrucciones que es necesario añadir para obtener los listados de las órbitas y la explicación textual.

❑ Paso 7: Describir las páginas del curso con *SODA*

Finalmente, hemos de construir los archivos con las descripciones *SODA* de las páginas del curso. En todas las páginas vamos a reutilizar dos índices y un pie de página. El pie de página es general a todos los cursos, y se describió en el listado VIII.2. El primer índice es un índice de todas las páginas del curso de gravitación, y es común a todas las páginas de este curso. El segundo es un índice de todos los cursos y es común a todas las páginas de todos los cursos. Ambos archivos definen una serie de tablas cuyos elementos son enlaces a las páginas apropiadas.

Como ejemplo, el listado X.4 muestra un esquema de la página del sistema Tierra-Luna, y la figura X.2 la página resulante.

```
* Tercera página del curso...
* AUTHOR Juan de Lara
* EMAIL  Juan.Lara@ii.uam.es
* DATE   22/12/99
TITLE The Moon: the Earth's satellite
DESCRIPTION From a point of view located on the Earth, the Moon revolves
DESCRIPTION around us with a period of about 27,5 days.\n
...
IMAGE [C], "../images/galmoon.gif" ,"A photograph of the Moon (by courtesy of NASA).
MODEL [600;650], [C], "luna.csm", "/tesis/courses"
INCLUDE "gravitacion\pdeindex.csm"
INCLUDE "coursesindex.csm"
INCLUDE "footnote1.csm"
```

Listado X.4: Un esquema del código necesario para generar la página del sistema Tierra-Luna

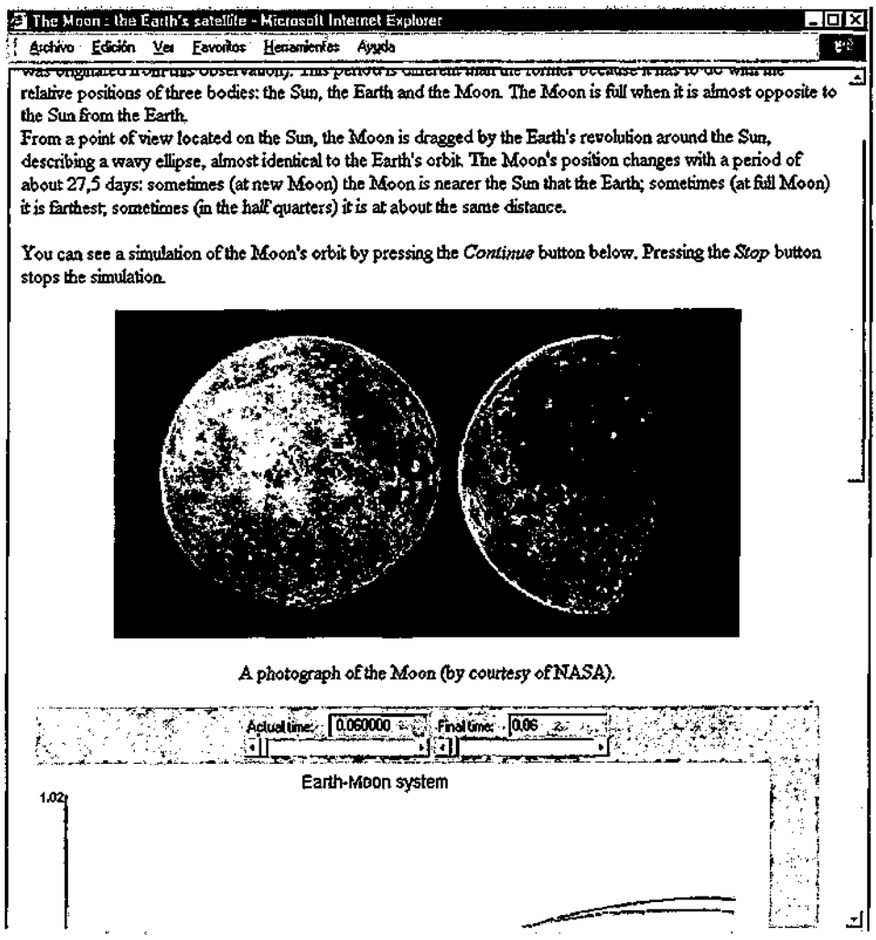


Figura X.2: Sistema Tierra-Luna.

■ X.3 Un curso sobre ecología

El curso sobre ecología [Alfo98g] [Alfo98c] [Alfo99d] se puede encontrar en la dirección:

http://www.ii.uam.es/~epulido/ecology/simul.htm

❑ Paso 1: Diseño del curso en papel

El curso consta de siete páginas en las que se incluyen simulaciones diversos ecosistemas:

- Un sistema aislado con tres especies (un productor primario, una presa y un predador), con los parámetros apropiados para mantenerlo en equilibrio oscilante. El estudiante puede observar la periodicidad de esta clase de sistemas.
- Un ecosistema con tres especies en equilibrio.
- Un sistema con tres especies, que inicialmente está en equilibrio, pero que es invadido por un predador, lo que hace que el sistema pierda el equilibrio. Transcurrido un cierto tiempo, el ecosistema llega a un estado de equilibrio oscilante.
- Una versión multimedia del modelo anterior, en la que se muestra la cadena trófica en cada momento, y explicaciones textuales en los momentos críticos de la simulación.
- El mismo sistema, pero invadido esta vez por un herbívoro. Se rompe el equilibrio, pero más tarde, el ecosistema también llega a un estado de equilibrio oscilante.

- Un ecosistema de cinco especies en equilibrio.
- Un ecosistema con cinco especies, pero con una interfaz en la que se deja al estudiante experimentar con el sistema, modificando los parámetros de la simulación. Alguno de estos experimentos son sugeridos en el texto de la página, pero el estudiante puede efectuar otros.
- Una simulación de la sabana africana simplificada, con quince especies diferentes que interaccionan creando cadenas tróficas complicadas y nichos ecológicos.
- Una simulación de tres ecosistemas, dos de ellos están formados por cuatro especies, el otro está formado por tres especies, y está en equilibrio. Se producen migraciones de un ecosistema a otro, debido bien a la época del año, o bien a un exceso de individuos de la misma especie. Se puede observar cómo una migración de cebras rompe el equilibrio del primer ecosistema.

❑ Paso 2: Construir los modelos necesarios

Los modelos se basan en una modificación de las ecuaciones de Volterra [Volt31], del modelo predador-presa, que contemplan los siguientes casos :

- Predadores que a su vez son presas.
- Especies que son presas de ciertos predadores, pero no de otros.
- Predadores omnívoros (se alimentan tanto de herbívoros como de productores primarios).
- Cadenas alimenticias de más de tres especies.
- Nichos ecológicos, especies vecinas que no pertencen a la misma cadena alimenticia, ni compiten por los mismos recursos.

Para nuestras ecuaciones, hemos considerado tres tipos diferentes de especies :

- Productores primarios, normalmente plantas, que son consumidos, pero no consumen. La evolución de su población *X* sigue la ecuación :

$$X' = m \cdot X - \Sigma\, n_i \cdot X \cdot Y_i$$

Ecuación X.1: Evolución de la población de productores primarios.

donde $[Y_i]$ es un vector con las poblaciones de las especies que consumen al productor primario. $[n_i]$ es un vector de coeficientes con el porcentaje de cada consumidor (herbívoro u omnívoro) que se alimenta del productor primario. *m* es el coeficiente de reproducción espontánea del productor primario, que regula la velocidad con la que la población aumenta en ausencia de consumidores.

- Superpredadores, que cazan, pero no son cazados. La evolución de su población *X* sigue la ecuación:

$$X' = -m \cdot X + \Sigma\, n_i \cdot X \cdot Y_i$$

Ecuación X.2: Evolución de la población de los superpredadores.

donde $[Y_i]$ es un vector con las poblaciones de las especies de las que se alimenta el superpredador, $[n_i]$ es un vector de coeficientes de apetencia, proporcional al porcentaje de cada presa en los hábitos alimenticios del superpredador. Finalmente *m* es el coeficiente de resistencia inversa al hambre del superpredador, que regula la velocidad con que la población disminuye en ausencia de comida.

- Consumidores intermedios, que cazan y son consumidos. La evolución de su población *X* viene dada por :

$$X' = -m \cdot X + \Sigma\, n_i \cdot X \cdot Y_i - \Sigma \cdot p_i \cdot X \cdot Z_i$$

Ecuación X.3: Evolución de la población de los superpredadores.

donde $[Y_i]$ es un vector con las poblaciones de las especies de las que se alimenta el consumidor, $[n_i]$ es un vector de coeficientes de apetencia. $[Z_i]$ es el vector de poblaciones de las especies que se alimentan del consumidor, $[p_i]$ es el vector de coeficientes correspondiente, y *m* es el coeficiente de resistencia inversa al hambre, que regula la velocidad con que la población disminuye en ausencia de comida y predadores.

En este modelo, no se hace distinción entre predadores y herbívoros. Un consumidor puede pertenecer a las dos categorías (como las especies omnívoras). El modelo está en equilibrio perfecto cuando las primeras derivadas de todas las especies son cero y no hay otros efectos, tales como epidemias o invasiones de especies.

En un modelo previo [Alfo98g], considerábamos los vectores $[n_i]$ y $[p_i]$ constantes para todas las especies. Pero esta suposición no se adaptaba a las situaciones reales. Por ejemplo, en un ecosistema en equilibrio con las siguientes cuatro especies :

- Un leon como superpredador.
- Cebras y ñus como consumidores.
- Hierba como productor primario.

El vector de apetencias para las cebras y los ñus es [0.7, 0.3], que quiere decir que en las condiciones iniciales, el 70% de las presas son ñus, y el 30% son cebras. Supongamos que los ñus sufren una epidemia y su población se reduce a la mitad. Si los coeficientes de apetencia de los leones fueran constantes, el número de cebras que el leon capturaría sería el mismo, pero cazaría menos ñus, y la población de leones decrecería inmediatamente. Por otra parte, la población de cebras aumentaría, porque encontrarían menos competencia para los productores primarios y sería menos capturada. Finalmente los ñus no se recobrarían nunca, porque sufrirían un porcentaje alto de predación por parte de los leones. El ecosistema sería inestable.

Esto no es lo que pasaría en un ecosistema real. Los leones se adaptarían al cambio modificando sus coeficientes de apetencia, incrementando la proporción de cebras y reduciendo la de ñus. El resultado sería un descenso inicial de la población de cebras (en lugar de un aumento) y enseguida un incremento, cuando la población de hierba aumentase, debido al decrecimiento de la población de ñus. Por otra parte, los ñus se recuperarían debido a que el coeficiente de apetencia de los leones decrecería. Después de un cierto tiempo, se obtendría una situación de equilibrio oscilante. El ecosistema sería estable.

Una forma de hacer esto en el modelo es hacer los coeficientes $[n_i]$ y $[p_i]$ proporcionales a las poblaciones. Este cambo resuelve el problema y lleva a situaciones estables, que se recuperan de pequeñas alteraciones.

Para implementar el modelo en *OOCSMP*, se ha construido una clase llamada *Species*, que contiene toda la información de una determinada especie, el listado X.5 muestra el código de la clase.

```
************************************************************************
* Definition of the Species class
************************************************************************
CLASS Species {
* ***************************** Data ********************************
 NAME name
 ICON icname
 DATA X0, M, N1, N2:=0, start:=0, max:=1000, K:=1
 DATA Il1:=10, When:=1000, Int:=1000
* ************************** Equations ******************************
 DYNAMIC
 X:=STEP(start)*LIMIT(0,max,XT)
 XT:=INTGRL(X0,XP)
 XP:=K*X*(M-IMPULS(When,Int)*Il1)
 XPdel1 :=0
 XPdel2 :=0
 TEats  :=0
 TEats0 :=0
* ************************** ACTION *********************************
 ACTION Species S,Percent, Last
  XPdel1 +=INSW(Percent, Percent*S.X*S.TEats0/S.TEats, 0)
  XPdel2 +=INSW(Percent, 0, Percent*S.X*S.X/S.X0)
  TEats  +=INSW(Percent, 0, 1)*S.X
  TEats0 +=INSW(Percent, 0, 1)*S.X0
  XP +=INSW(Percent, Last*K*N2*X*XPdel1*X/X0, Last*K*N1*X*XPdel2*TEats0/TEats)
}
```

Listado X.5: clase *Species*.

Cada especie contiene los siguientes atributos y parámetros :

- *XO* : la población inicial.
- *X* : la población actual.
- *XP* : Primera derivada de la población.
- *M, N1, N2* : Coeficientes en las ecuaciones de Volterra extendidas.
- *start* : Valor del tiempo en el que la especie invade el ecosistema (su valor por defecto es cero, es decir, que la especie es un miembro permanente del ecosistema).
- *Ill* : Intensidad de una posible epidemia que efecte a la especie.
- *When* : Valor del tiempo en que se produce la primera aparición de la epidemia.
- *Int* : Intervalo de repetición de la epidemia.

Los valores por defecto de los tres últimos parámetros se han configurado para que no ocurran epidemias.

Para la última página del curso, en la cual se incluye una simulación de ecosistemas con migraciones, hemos modificado la clase *Species* (ver apartado VI.3.2).

❑ Paso 3: Adaptar el modelo para cada página

Si bien la clase *Species* es la misma para las seis primeras páginas del curso, se ha tenido que modificar los coeficientes para que se adapten a cada especie y situación particular. Además, en cada modelo se ha tenido que crear un número de especies diferentes.

- En la primera página hay un ecosistema de tres especies (León-Ñu-Hierba), que no está en equilibrio. La figura X.3, muestra la cadena alimenticia.

- La segunda página contiene el mismo ecosistema de la página anterior, pero en equilibrio. Hay que calcular los coeficientes de equilibrio, haciendo que las derivadas (que son la variaciones de las poblaciones) sean iguales a cero.

- En la tercera página, en principio hay un ecosistema en equilibrio de tres especies, como en las dos anteriores, pero en *t=50*, se produce la entrada de un predador (leopardo), que rompe el equilibrio. La figura X.4 muestra la cadena trófica.

- En la cuarta página, el modelo es el mismo, si bien se añadirán elementos multimedia.

- En la quinta página, en principio hay un ecosistema en equilibrio de tres especies, como en las tres anteriores, pero en *t=50*, se produce la entrada de un herbívoro (cebra), que rompe el equilibrio. La figura X.5 muestra la cadena trófica.

- En la sexta página, en principio hay un ecosistema en equilibrio de cinco especies (leon, ñu, cebra dos tipos de hierba). La figura X.6 muestra la cadena trófica.

- La séptima página es una página de experimentación libre con el ecosistema de cinco especies. Se proponen varios experimentos al alumno, que consisten en modificaciones de los parámetros, tales como cambiar los parámetros principales (*M*) de cada especie, cambiar las poblaciones iniciales, cambiar el momento de entrada para alguna de las especies, producir epidemias en alguna especie, si bien el alumno puede experimentar libremente con la página.

- En la página de la sabana africana, los datos de los parámetros y de los coeficientes son más o menos reales, se han sacado de la literatura [Rodr70]. Este modelo contiene quince especies, cuatro de los cuales son carnívoros, seis son herbívoros, dos omnívoros, y tres productores primarios (plantas). Hay varias cadenas alimenticias, la más grande con cinco especies: leon-chacal-rata-langosta-hierba. El listado X.6 muestra un esquema del modelo *OOCSMP* de la sabana africana..

- En la última página, de los distintos ecosistemas con migraciones, el primer ecosistema está formado por tres especies en equilibrio (leon-ñu-hierba), el segundo y tercer ecosistemas están formados por cuatro especies (leon-cebra-ñu-hierba) que no están en equilibrio. Este modelo es similar al del apartado VI.3.2, pero sin el esquema de distribución.

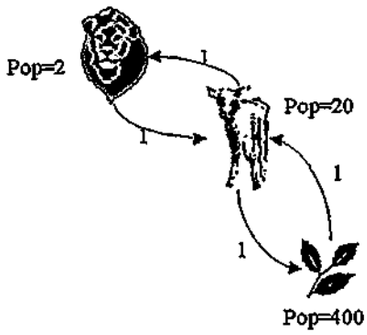


Figura X.3 : Cadena alimenticia de las dos primeras páginas.

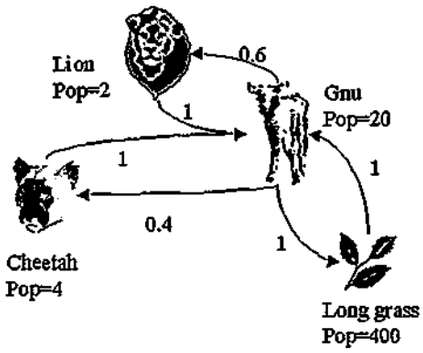


Figura X.4 : Cadena alimenticia de la página tres y cuatro.

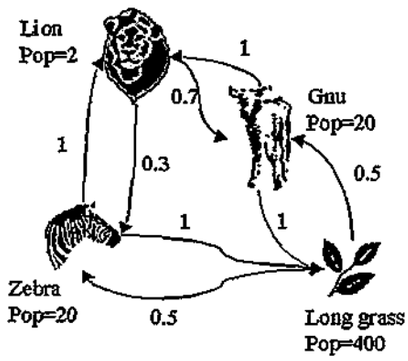


Figura X.5: Cadena alimenticia de la página cinco.

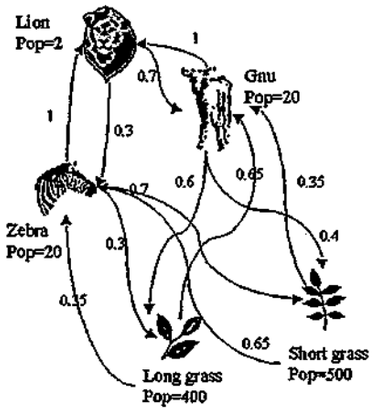


Figura X.6: Cadena alimenticia de la página seis.

```
TITLE Savanna
INCLUDE "Species.csm"
*****************************************************************
* Actual species                                                *
*****************************************************************
Species Lion    ("Lion",    "icons/lion002.gif"   , 2,=,0195,.001          )
Species Cheetah("Cheet",    "icons/wcat002.gif"   ,4,-.028, .001           )
Species Hyaena  ("Hyaen",   "icons/wanim001.gif"  ,4,-.024, .001           )
Species Jackal  ("Jacka",   "icons/wolf001.gif" ,10,-.0128,.0002,  .0012)
Species Zebra   ("Zebra",   "icons/zebra003.gif",20,-.012, .0001,  .01   )
Species Gnu     ("Gnu",     "icons/bovin008.gif",20,-.01,  .0001,  .01   )
Species Buffalo("Buffa",    "icons/buffs001.gif",10,-.02,  .00007, .004  )
Species Gazelle("Gazel",    "icons/wanim012.gif",30,-.025, .000115,.005  )
Species Giraffe("Giraf",    "icons/giras001.gif",10,-.01,  .00028, .002  )
Species Boar    ("Boar",    "icons/pig002.gif",  30,-.021, .00025, .0125)
Species Rat     ("Rat",     "icons/wanim010.gif",80,-.01,  .00015, .002  )
Species Locust  ("Locus",   "icons/insec050.gif",100,-.02,  .000305,.002 )
Species LGrass  ("LGras",   "icons/leafs015.gif",400, .016, 0,      .0005)
Species SGrass  ("SGras",   "icons/leafs029.gif",400, .027, 0,      .0005)
Species Acacia  ("Acaci",   "icons/leafs013.gif",50, .01,  0,      .001  )
Species EcoSystem:=   Lion,Cheetah,Hyaena,Jackal,Zebra,Gnu,Buffalo,
                      Gazelle,Giraffe,Boar,Rat,Locust,LGrass,SGrass,Acacia
DYNAMIC
      EcoSystem.STEP()
      Lion.ACTION(Gnu,     0.3,  0)
      Lion.ACTION(Zebra,   0.25, 0)
      Lion.ACTION(Buffalo, 0.15, 0)
      Lion.ACTION(Gazelle, 0.1,  0)
      Lion.ACTION(Boar,    0.1,  0)
      Lion.ACTION(Giraffe, 0.06, 0)
      Lion.ACTION(Jackal,  0.04, 1)
```

```
        Cheetah.ACTION(Gazelle, 0.65, 0)
        Cheetah.ACTION(Gnu,     0.2,  0)
        Cheetah.ACTION(Boar,    0.15, 1)
        Hyaena.ACTION(Gnu,      0.4,  0)
        Hyaena.ACTION(Gazelle,  0.4,  0)
        Hyaena.ACTION(Zebra,    0.2,  1)
        Jackal.ACTION(Rat,      0.5,  0)
        Jackal.ACTION(Locust,   0.3,  0)
        Jackal.ACTION(Gazelle,  0.2,  1)
        Jackal.ACTION(Lion,     -1,   1)
        Zebra.ACTION(LGrass,    1,    1)
        Zebra.ACTION(Lion,      -0.6, 0)
        Zebra.ACTION(Hyaena,    -0.4, 1)
        Gnu.ACTION(LGrass,   1,    1)
        Gnu.ACTION(Lion,     -0.5, 0)
        Gnu.ACTION(Cheetah,  -0.3, 0)
        Gnu.ACTION(Hyaena,   -0.2, 1)
        Buffalo.ACTION(LGrass,   1,    1)
        Buffalo.ACTION(Lion,     -1,   1)
        Gazelle.ACTION(SGrass,   1,    1)
        Gazelle.ACTION(Cheetah,  -0.5, 0)
        Gazelle.ACTION(Lion,     -0.2, 0)
        Gazelle.ACTION(Hyaena,   -0.2, 0)
        Gazelle.ACTION(Jackal,   -0.1, 1)
        Giraffe.ACTION(Acacia,   1,    1)
        Giraffe.ACTION(Lion,     -1,   1)
        Boar.ACTION(SGrass,   0.5,  0)
        Boar.ACTION(Rat,      0.3,  0)
        Boar.ACTION(Locust,   0.2,  1)
        Boar.ACTION(Cheetah,  -0.6, 0)
        Boar.ACTION(Lion,     -0.4, 1)
        Rat.ACTION(Locust,    0.4,  0)
        Rat.ACTION(SGrass,    0.3,  0)
        Rat.ACTION(LGrass,    0.3,  1)
        Rat.ACTION(Jackal,    -0.7, 0)
        Rat.ACTION(Boar,      -0.3, 1)
        Locust.ACTION(SGrass,   0.5,  0)
        Locust.ACTION(LGrass,   0.5,  1)
        Locust.ACTION(Rat,      -0.5, 0)
        Locust.ACTION(Boar,     -0.3, 0)
        Locust.ACTION(Jackal,   -0.2, 1)
        LGrass.ACTION(Zebra,    -0.3, 0)
        LGrass.ACTION(Gnu,      -0.3, 0)
        LGrass.ACTION(Buffalo,  -0.2, 0)
        LGrass.ACTION(Rat,      -0.1, 0)
        LGrass.ACTION(Locust,   -0.1, 1)
        SGrass.ACTION(Gazelle,  -0.4, 0)
        SGrass.ACTION(Boar,     -0.2, 0)
        SGrass.ACTION(Rat,      -0.2, 0)
        SGrass.ACTION(Locust,   -0.2, 1)
        Acacia.ACTION(Giraffe,  -1,   1)
* Timer data
TIMER delta:=0.1,FINTIM:=900,PRdelta:=10,PLdelta:=5
METHOD ADAMS
```

Listado X.6 : Ecosistema de la sabana africana.

❑ Paso 4: Validar el modelo

El modelo se ha validado experimentalmente. Ver la discusión del paso 2.

❑ Paso 5: Decisión de las salidas gráficas

En todas las páginas vamos a incluir en la parte superior del *applet* un gráfico 2D con la evolución de las poblaciones. En la parte inferior vamos a incluir un gráfico icónico también con la evolución de las poblaciones, donde cada icono va a representar un individuo o un porcentaje de individuos de una especie.

En la página de la sabana africana, además, vamos a incluir botones que presenten diversas cadenas alimenticias por separado en el gráfico 2D. Esto lo vamos a lograr con la instrucción '\'. Se van a presentar las siguientes cadenas:

- Leon, Chacal, Rata, Langosta y Hierba corta.
- Leon, girafa y Acacia.
- Leopardo, Jabalí, Rata, Langosta, y Hierba corta.

Para realizar esto, son necesarias las siguientes instrucciones :

```
PLOT  [C], Lion.X, Jackal.X, Rat.X, Locust.X, SGrass.X, TIME
ICONICPLOT [S], Lion.X, Cheetah.X, Hyaena.X, Jackal.X, Zebra.X, Gnu.X, Buffalo.X,
                Gazelle.X,Giraffe.X,Boar.X,Rat.X,Locust.X,LGrass.X,SGrass.X,Acacia.X
\
TITLE Chain 2
PLOT [C], Lion.X, Giraffe.X, Acacia.X, TIME
\
TITLE Chain 3
PLOT [C], Cheetah.X, Boar.X, Rat.X, Locust.X, SGrass.X, TIME
```

Listado X.7: Añadiendo salidas gráficas al ecosistema de la sabana africana.

❑ Paso 6: Inclusión de los elementos multimedia

Sólamente vamos a añadir elementos multimedia a la página cuatro. Este paso se realizó en el apartado VI.3.1.

❑ Paso 7: Describir las páginas del curso con *SODA*

Finalmente, hemos de construir los archivos con las descripciones *SODA* de las páginas del curso. En todas las páginas vamos a reutilizar dos índices y un pie de página. El pie de página es general a todos los cursos, y se describió en el listado VIII.2. El primer índice es un índice de todas las páginas del curso de ecología, y es común a todas las páginas de este curso. El segundo es un índice de todos los cursos y es común a todas las páginas de todos los cursos. Ambos archivos definen una serie de tablas cuyos elementos son enlaces a las páginas apropiadas.

En las páginas dos, tres, cuatro, cinco y seis incluiremos una imagen con la cadena trófica. Como ejemplo, la siguiente figura muestra la página del ecosistema invadido por un herbívoro.

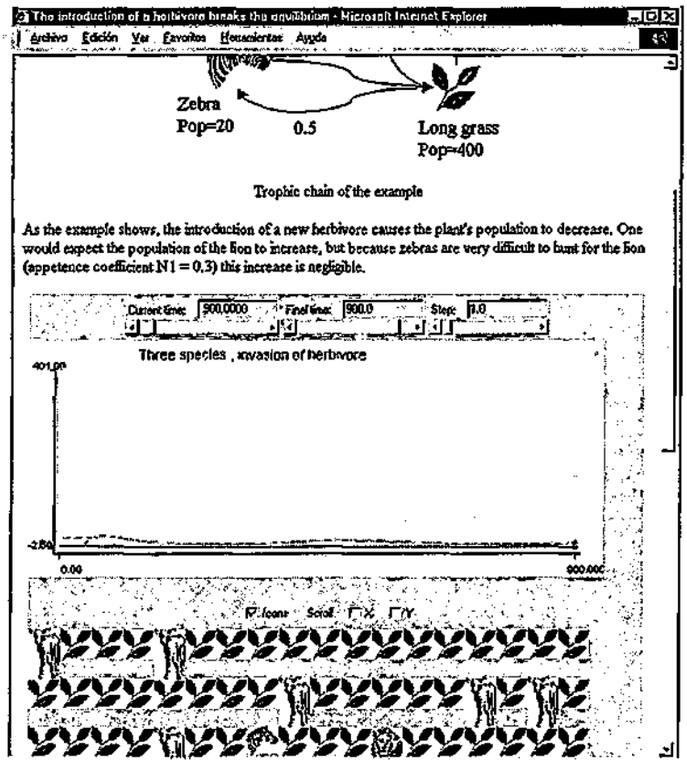


Figura X.7: Página del ecosistema invadido por un herbívoro.

X.4 Un curso sobre electrónica básica

Este curso [Alfo98d], [Alfo00c] se puede encontrar en:

http://www.ii.uam.es/~epulido/circ/modules.htm

Paso 1: Diseño del curso en papel

Vamos a dividir el curso en dos partes:

- Circuitos con lógica combinacional.
- Circuitos secuenciales.

En la parte de circuitos de lógica combinacional, vamos a presentar páginas para la construcción de multiplexores, decodificadores y sumadores. Para construir estos tres elementos, se presenta una primera página en la que se construye un circuito simple (multiplexores 2x1, decodificadores 2x4, y sumadores de un bit). También se ha añadido una página que simula el funcionamiento de una *PAL* 4x4 (Programmable Array Logic). En páginas posteriores se muestra el proceso de construcción de elementos más complicados a partir de estos elementos simples. En la parte de circuitos secuenciales se ha implementado un biestable de tipo D. Es decir, el esquema de las páginas quedaría como sigue :

- Introducción a los sistemas digitales.
  - Circuitos combinacionales
    - Multiplexores 2x1
      - Multiplexores 4x1
    - Decodificadores 2x4
      - Decodificadores 4x16 (en árbol y concurrentes)
    - Sumadores de 1 bit
      - Sumadores de cuatro bits.
    - *PAL 4x4.*
  - Circuitos secuenciales, biestable tipo D.

Paso 2 : Construir los modelos necesarios

Para la construcción de este curso, vamos a necesitar modelos *OOCSMP* de multiplexores (2x1 y 4x1), Decodificadores (2x4 y 4x16), Sumadores (1 y 4 bits) y *PALs*. Para ello, primero se construyeron los modelos de los elementos más sencillos. Una vez hecho esto, se encapsularon en clases *OOCSMP*, y se construyeron los elementos más complejos utilizando estas clases. La construcción de este curso dio pie a la construcción de una librería *OOCSMP* de componentes electrónicos, que será detallada en el apéndice 'A'. Como ejemplo, vamos a construir los modelos necesarios para la página de los multiplexores, para el resto de las páginas, se han seguido procedimientos muy similares.

El siguiente listado, muestra el código *OOCSMP* del modelo del multiplexor 2x1.

```
TITLE Multiplexor 2x1
DATA X0:=0,X1:=0,S:=0,E:=1
DYNAMIC
  notS := NOT (S)
  and1 := AND ( X0, notS )
  and2 := AND ( X1, S )
  Yp := IOR ( and1, and2 )
  Y  := AND ( E , Yp )
TIMER delta:=1, FINTIM:=1, PRdelta := 1
```

Listado X.8: Modelo de un Multiplexor.

Ahora hemos de encapsular el multiplexor en una clase *OOCSMP* (listado X.9).

```
* TITLE Esta clase implementa un multiplexor de 2x1
* AUTHOR Juan de Lara Jaramillo
* EMAIL Juan.Lara@ii.uam.es
CLASS MUX2x1
(
NAME name
ICON icname = "add1.gif"
* Las entradas del multiplexor : X0,X1,S,E
* (E habilita el multiplexor )
DYNAMIC X0,X1,S,E
  notS := NOT (S)
  and1 := AND ( X0, notS )
  and2 := AND ( X1, S )
  Yp := IOR ( and1, and2 )
  Y  := AND ( E , Yp )
* La salida del multiplexor : Y
)
```

Listado X.9: Modelo encapsulado de un Multiplexor 2x1.

Nótese que, como la sección *DYNAMIC* no puede devolver ningún valor, el resultado de la clase anterior se ha dejado en el atributo *Y*. Actualmente estamos recodificando estos elementos, de forma que no incluimos el código en la sección *DYNAMIC*, sino en un método, que sí devuelve un valor, en lugar de dejarlo en un atributo. Además el curso también está siendo extendido [Alfo00c].

La clase anterior toma como icono por defecto el fichero *add1.gif*. Ha habido que hacer un dibujo para cada componente de la biblioteca, así como para cada bloque estándar *OOCSMP*.

Para acabar con los multiplexores, se va a construir un modelo de un multiplexor 4x1 utilizando la clase del listado X.9. La salida queda almacenada en la variable *Y*.

```
DATA X0 := 0, X1 := 0, X2 := 0, X3 := 0
DATA S0 := 0, S1 := 0, Y  := 0, E:=1
INCLUDE "circ\MUX2x1.CSM"
MUX2x1 m1 ("m1")
MUX2x1 m2 ("m2")
MUX2x1 m3 ("m3")
DYNAMIC
  notS0 := NOT(S0)
  m1.STEP(X0,X1,S0,E)
  m2.STEP(X2,X3,S0,E)
  m3.STEP(m1.Y,m2.Y,S1,E)
  Y=m3.Y
TIMER delta:=1, FINTIM:=1
```

Listado X.10: Modelo de un Multiplexor 4x1.

❑ Paso 3 : Adaptar el modelo para cada página

En este paso vamos a programar un modelo que use cada componente. Para ello, basta con conectar una entrada al componente, y guardar la salida en una variable.

Siguiendo con el ejemplo del multiplexor, vamos a preparar un modelo (listado X.11) que use la clase *MUX2x1* y lo colocaremos en la primera página junto con el modelo del listado X.8.

```
TITLE Multiplexor 2x1
INCLUDE "circ\MUX2x1.CSM"
DATA X0:=0,X1:=0,S:=0,E:=1
MUX2x1 m1 ("m1")
DYNAMIC
   m1.STEP(X0,X1,S,E)
TIMER delta:=1, FINTIM:=1
CONNECTIONPLOT
```

Listado X.11: Modelo que usa la clase del listado X.9.

❑ Paso 4 : Validar los modelos

Para validar los modelos, se han probado con todas las entradas posibles, y se ha comparado con las tablas que se pueden encontrar en la literatura [Floy97]. Para probar con todas las entradas, se han modificado los modelos para que la entrada sea una onda en la que se generan todas las combinaciones posibles de impulsos 0 ó 1, finalmente se imprime el resultado, y se compara con la tabla. Por ejemplo, el siguiente listado (X.12) muestra el procedimiento de validación para el multiplexor 4x1. En el resto de modelos se ha actuado de manera similar.

```
INCLUDE "circ\MUX41.CSM"
DATA X[4], Sel[2]
MUX41 m("m")
DYNAMIC
       X[0] := IMPULS ( 1,4 )
       X[1] := IMPULS ( 2,4 )
       X[2] := IMPULS ( 3,4 )
       X[3] := IMPULS ( 4,4 )
       Sel[0]:= IMPULS ( 1,2 )
       Sel[1]:= IMPULS ( 2,4 )+ IMPULS( 3,4 )
       m.STEP(X,Sel,1)
* Parametros de impresión y de tiempo...
PRINT Sel, X, m.Y
TIMER delta := 1, PRdelta := 1, FINTIM := 7
```

Listado X.12: Validación de un multiplexor 4x1.

❑ Paso 5: Decisión de las salidas gráficas

En todas las páginas, se va a incluir un gráfico con la circuitería del circuito que estamos simulando. Esto se logra mediante la instrucción *CONNECTIONPLOT*.

❑ Paso 6: Inclusión de los elementos multimedia

En este curso no vamos a incluir elementos multimedia.

❑ Paso 7 : Describir las páginas del curso con *SODA*

Finalmente, hemos de construir los archivos con las descripciones *SODA* de las páginas del curso. En todas las páginas vamos a reutilizar un índice y un pie de página. El pie de página es general a todos los cursos, y se describió en el listado VIII.2. El índice es el de todos los cursos y es común a todas las páginas de todos los cursos. Ambos archivos definen una serie de tablas cuyos elementos son enlaces a las páginas apropiadas. Como ejemplo, se muestra un esquema de la descripción de la página en la que se encuentra el multiplexor más básico.

```
* Cuarta página del curso de electronica
* AUTHOR Juan de Lara
* EMAIL  Juan.Lara@ii.uam.es
* DATE   1/1/2000
TITLE Multiplexers
DESCRIPTION A multiplexer circuit accepts N inputs and outputs the value
DESCRIPTION of one of those inputs.\n
...
DESCRIPTION The I/O for a multiplexer is shown in the following figure.\n
IMAGE [C], "../images/mul.jpg", "I/O for the multiplexer"
DESCRIPTION The truth table for a 2-to-1 multiplexer is shown in the following table:\n
TABLE [9;4], [C,50],
      "Control", "Input0", "Input1", "Output",
      "0",       "0",      "0",      "0",
      "0",       "0",      "1",      "0",
      "0",       "1",      "0",      "1",
      "0",       "1",      "1",      "1",
      "1",       "0",      "0",      "0",
      "1",       "0",      "1",      "1",
      "1",       "1",      "0",      "0",
      "1",       "1",      "1",      "1",
```

```
      "Truth table for the multiplexer"
...
DESCRIPTION These are the instructions to play with the circuit:\n
DESCRIPTION \ITEM{Click on the 1's and 0's in green (the inputs) to change the
DESCRIPTION value. To run the simulation with these new values, click on the "
DESCRIPTION value.}
DESCRIPTION \ITEM{Click on any of the logic gates and its output value will ap
DESCRIPTION "Block value" box.}
DESCRIPTION \ITEM{The "Reset" button will clear all the input values.}
MODEL [270;450], [C], "mux_2.csm", "/tesis/courses"
DESCRIPTION This circuit can be encapsulated in a single module, as can be see
DESCRIPTION the following figure :
MODEL [380;380], [C], "mux_1.csm", "/tesis/courses"
DESCRIPTION A 4 - bit multiplexer circuit is constructed in the
DESCRIPTION \LINK{"4mult.html" next page}
INCLUDE "coursesindex.csm"
INCLUDE "footnote1.csm"
```

Listado X.13: Página del multiplexor más sencillo.

Una vez compilado, se obtiene la siguiente página:

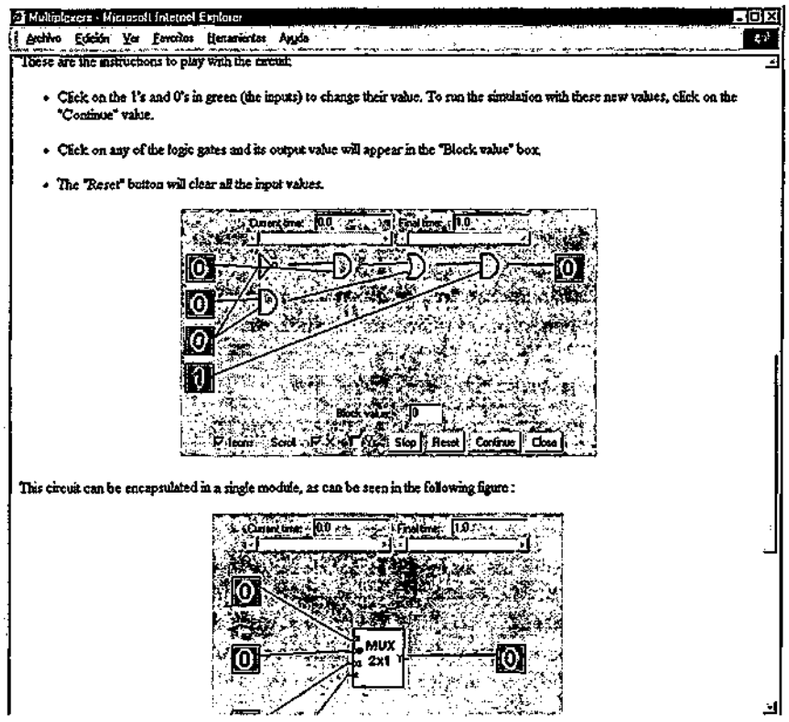


Figura X.8: Página del multiplexor más sencillo.

X.5 Un curso sobre ecuaciones en derivadas parciales

Este curso [Alfo99d] puede encontrarse en la dirección Internet:

http://www.ii.uam.es/~jlara/investigacion/index.html

Paso 1: Diseño del curso en papel

Este curso está dividido en tres grupos de páginas:

- El primer grupo, está formado por cuatro páginas de teoría. La primera introduce al alumno en los conceptos básicos de las ecuaciones en derivadas parciales. Las dos siguientes presentan una introducción al método de los elementos finitos. La cuarta es una introducción al método de las diferencias finitas.
- El segundo grupo está formado por páginas que contienen simulaciones de la solución de ecuaciones *típicas, tales como* :
  - Ecuación del calor en una dimensión (dependiente del tiempo) con distintos coeficientes de conducción en distintas partes de la barra. Este problema fue planteado en el apartado V.6.2.
  - La ecuación del transporte no difusivo en una dimensión, dependiente del tiempo.
  - La ecuación del transporte difusivo en una dimensión, dependiente del tiempo.
  - La ecuación del calor en dos dimensiones no dependiente del tiempo.
  - La ecuación del calor en dos dimensiones, dependiente del tiempo, resuelta mediante elementos finitos y mediante un autómata celular.
  - Un ejemplo de generación de mallas simple. Se discretizarán las primitivas básicas en dos dimensiones, utilizando generadores de mallas de tipo Delaunay, y de tipo isoparamétrico. También se discretizarán dominios un poco más complicados, tales como una pala y un arco. Esta discretización se llevará a cabo mezclando las dos anteriores formas de generación.
  - Una página con mallas móviles, que muestra ejemplos de aplicación de operaciones geométricas a diversas mallas en el bucle principal de la simulación.
- El tercer grupo está formado por páginas que contienen aplicaciones más complejas, como:
  - El calentamiento de varias vigas, peresentado en la sección V.6.1.
  - El calentamiento de dos vigas móviles, presentado en la sección VII.3.2
  - La solución de una ecuación de difusión (*Ut+Uxx+Uxy+Uyy = 0*), presentado en la sección VI.2.4.
  - El mismo problema, pero diseñando la geometría con *MGEN*, presentado en la sección VI.2.4.
  - Ecuación del calor en una dimensión, combinando varias salidas gráficas (problema de la sección VI.2.3).

Paso 2: Construcción de los modelos necesarios

- Para la primera página, se ha de construir un modelo de la ecuación del calor en una dimensión. Para ello vamos a usar el modelo del apartado V.6.2.

- Para la segunda página, se ha de construir un modelo de la ecuación del transporte no difusivo en una dimensión, que viene dada por la expresión :

$$\frac{\partial F}{\partial t} + u\frac{\partial F}{\partial x} = 0$$

Ecuación X.4: Ecuación del transporte no difusivo en una dimensión.

En está ecuación, u es la velocidad de propagación. Se quiere transportar una onda senoidal de izquierda a derecha. El método de resolución que usaremos será de diferencias finitas, con esquema de DuFort-Frankel. Hay que tener en cuenta que el número de Courant viene dado por u/(Δx/Δt), para que la solución que obtenemos sea no difusiva realmente, la expresión anterior debe ser igual a 1. Por ejemplo, si elegimos la distancia entre dos puntos de la malla discretizada igual a 0.1, y el incremento de tiempo igual a 0.001, la velocidad de propagación debe ser 100 para obtener una solución exacta. Estos van a ser

los valores que vamos a asignar por defecto al modelo. El alumno puede cambiarlos para ver lo que pasa con distintos valores del número de Courant.

El siguiente modelo (listado X.14) muestra cómo quedaría el modelo *OOCSMP*. En comentarios se ha incluido en cada línea el paso de la figura V.10 que se completa.

```
TITLE Non-diffusive transport in 1d
DATA u:=100
DOMAIN b1d := BAR (0.0, 5.0,                                    * PASO 1
                   ESSENTIAL(EDGE(1),SIN(delta*100)))           * PASO 2
MESH m1d:= ISOPARAMETRIC (b1d, LINE2, ELEMENTS(50))             * PASO 3
PDE trNoDiff1D (0,0,1,0,0,0,0,0,0,0,0,u,0,0,0,DUFORT)           * PASO 5
m1d.setPDE(trNoDiff1D)                                          * PASO 6
DYNAMIC
  m1d.STEP()                                                    * PASO 7
TIMER FINTIM:=0.05, delta:=0.001
PLOT3D [C], m1d
```

Listado X.14: Modelo del transporte no difusivo en una dimensión.

También hemos encapsulado el ejemplo anterior en una clase, que se puede parametrizar con :

- Velocidad de transporte.
- Coordenadas de inicio y fin de la barra.
- Número de nodos.

y la hemos añadido a la librería *OOCSMP* de *PDEs*. Hemos construido una clase similar con cada método de resolución (*EXPLICIT*, *IMPLICIT*, *FEM*), que han dado lugar a cinco elementos :

- *trNodiff_B1D_ISO_L2_EX* (diferencias finitas explícito).
- *trNodiff_B1D_ISO_L2_IM* (diferencias finitas implícito).
- *trNodiff_B1D_ISO_L2_FE* (elementos finitos lineal, símplice *LINE2*).
- *trNodiff_B1D_ISO_L3_FE* (elementos finitos cuadrático, símplice *LINE3*).

Es de hacer notar, que se pueden concatenar elementos de este tipo y resolver la ecuación mediante distintos métodos de resolución en cada objeto (ver problema de la sección V.6.3). Obviamente, si no queremos una ecuación con velocidad constante, estos elementos no nos valdrían, habría que programar otros nuevos, con la expresión *OOCSMP* requerida en lugar de u. El modelo de la clase se muestra en el listado X.19.

```
CLASS trNoDiff_B1D_ISO_L2_DF
{
 * TITLE Elemento barra unidimensional para el transporte
 ** no difusivo, mediante el método de Du Forte-Frankel
 * AUTHOR Juan de Lara
 * EMAIL Juan.Lara@ii.uam.es
 NAME name
 DATA xInit:=0, xFin:=0, u := 100, ne:=50
 ICON icname := "barTrp.gif"
 DOMAIN b1d := BAR(xInit,xFin)
 MESH m1d:=ISOPARAMETRIC(b1d,LINE2,ELEMENTS(ne))
 PDE trNoDiff1D (0,0,1,0,0,0,0,0,0,0,0,u,0,0,0,DUFORT)
 m1d.setPDE(trNoDiff1D)
 DYNAMIC
   m1d.STEP()
}
```

Listado X.15: Elemento unidimensional para el transporte difusivo.

Se ha añadido el atributo de icono, por si se elige un gráfico *CONNECTIONPLOT* como salida. Además, no se han impuesto condiciones de contorno, se deberán especificar después de la creación del objeto, con los métodos *ESSENTIAL*, o *NATURAL*. Se puede encontrar más información sobre los elementos de la biblioteca estándar de *OOCSMP* en el apéndice 'A'.

- Para la tercera página, se ha de construir un modelo de la ecuación del transporte difusivo en una dimensión, que viene dada por la expresión :

$$\frac{\partial F}{\partial t}+u\frac{\partial F}{\partial x}-\frac{\partial}{\partial x}\left(K\frac{\partial F}{\partial x}\right)=0$$

Ecuación X.5: Ecuación del transporte difusivo en una dimensión.

En este ejemplo, también vamos a transportar una onda sinoidal. El modelo (listado X.16) es muy similar al del listado X.14, excepto en la declaración de la *PDE*, en la que se ha introducido un coeficiente de difusión, *K*. También vamos a encapsular el transporte difusivo (ver apéndice 'A'), de forma parecida que al ejemplo anterior, sólo que ahora las clases tienen un parámetro más (la constante de difusión).

```
TITLE Diffusive transport in 1d
DATA u:=100, K:=1
DOMAIN b1d := BAR (0.0, 5.0,                                    * PASO 1
                   ESSENTIAL(EDGE(1),SIN(delta*100)))           * PASO 2
MESH m1d:= ISOPARAMETRIC (b1d, LINE2, ELEMENTS(50))             * PASO 3
PDE trDiff1D (0,0,1,-1,K,0,0,0,0,0,0,u,0,0,0,DUFORT)            * PASO 5
m1d.setPDE(trNoDiff1D)                                          * PASO 6
DYNAMIC
  m1d.STEP()                                                    * PASO 7
TIMER FINTIM:=0.05, delta:=0.001
```

Listado X.16: Modelo del transporte difusivo en una dimensión.

- Para la cuarta página, se ha de construir un modelo de la ecuación del transporte no difusivo en dos dimensiones, que viene dada por la expresión :

$$\frac{\partial F}{\partial t}+a\frac{\partial F}{\partial x}+b\frac{\partial F}{\partial y}=0$$

Ecuación X.6: Ecuación del transporte no difusivo en dos dimensión.

donde *a* y *b* son la velocidad de propagación en la dirección x e y respectivamente. Esta ecuación tiene como solución exacta : $u(t,x,y)=u_0(x-at,\ y-bt)$. Donde u_0 es la condición inicial que hemos impuesto (la función que vamos a transportar). En el presente problema, estableceremos como condición inicial, un cono de altura 1 en el centro del cuadrado, que es donde vamos a resolver la ecuación (en un cuadrado de lado dos y esquina inferior izquierda en -1,-1). Como condiciones de borde, y suponiendo que *a* y *b* son positivas (transporte hacia la derecha y hacia arriba), vamos establecer en los bordes inferior e izquierdo (1 y 4) la solución exacta. El modelo también realiza el cálculo de la solución exacta en cada iteración y lo almacena en una matriz.

```
TITLE Transport Equation in 2D
DATA EXACT[60;60]
DATA A:=1, B:=2, DX := 60, DY:= 60, LADO := 2
DOMAIN rectD := QUADRILATERAL ( -1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, 1.0,
              INITIAL(INSW(ABS(X)-0.5,INSW(ABS(Y)-0.5,(1-2*ABS(X))* (1-2*ABS(Y)),0),0)),
              ESSENTIAL(EDGE(1,4),
                      INSW(   ABS(X-A*TIME)-0.5,INSW(ABS(Y-B*TIME)-0.5,
                              (1-2*ABS(X-A*TIME))* (1-2*ABS(Y-B*TIME)),0),0)) )

MESH rectSol := ISOPARAMETRIC ( rectD, QUADRILAT4, ELEMENTS(60,60) )

PDE  tr2D ( 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, A, B, 0, 0, DUFORTE )

rectSol.SETPDE(tr2D)

DYNAMIC
  EXACT[i;j] := INSW( ABS(j*LADO/DX-A*TIME)-0.5,INSW(ABS(i*LADO/DY-B*TIME)-0.5,
                      (1-2*ABS(j*LADO/DX-A*TIME))* (1-2*ABS(i*LADO/DY-B*TIME)),0),0))
  rectSol.STEP()
TIMER delta := 0.0025, PLdelta:= 40*delta, FINTIM := 0.6
```

Listado X.17: Modelo del transporte no difusivo en dos dimensiones.

Al igual que en anteriores problemas, se ha encapsulado esta ecuación en una clase. También se ha encapsulando esta misma ecuación con todas las primitivas 2D de construcción de dominios de *OOCSMP*, y con todos los métodos de resolución disponibles, con mallado isoparamétrico y triangulación, con todos los símplices posibles. El ejemplo quedaría encapsulado de la siguiente forma :

```
CLASS trNodiff_Q42D_ISO_Q4_DF
{
    * TITLE Elemento cuadrilátero bidimensional para transporte
    * ABSTRACT Elemento cuadrilátero bidimensional con
    ** esquina inferior izquierda en (x1,y1) de dimensiones sx x sy
    ** con una rotación angrot. Discretizado con método
    ** isoparamétrico (DX x DY), símplice cuadrilátero de 4 nodos.
    * AUTHOR Juan de Lara
    * EMAIL Juan.Lara@ii.uam.es
    NAME name
    DATA A := 1, B:=2, DX := 50, DY := 50
    DATA x1:=0, y1:=0, sx:=1, sy:=1, angrot:=0
    ICON icname := "quadTrp.gif"
    DOMAIN rectD := QUADRILATERAL(x1,y1,x1+sx,y1,x1+sx,y1+sy,x1,y1+sy)
    rectD.ROTATE(angrot)
    MESH rectSol := ISOPARAMETRIC ( rectD, QUADRILAT4, ELEMENTS(DX,DY) )
    PDE  tr2D ( 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, A, B, 0, 0, DUFORTE )
    rectSol.SETPDE(tr2D)
    DYNAMIC
      m1d.STEP()

}
```

Listado X.18: Elemento *trNoDiff_Q42D_ISO_Q4_DF*.

Se puede encontrar más información sobre los distintos elementos en el apéndice A.

- Para la quinta página, se ha de construir un modelo de la ecuación del calor estática (no dependiente del tiempo) en dos dimensiones, que viene dada por la expresión :

$$\frac{\partial F}{\partial x}\left(-Kx\frac{\partial F}{\partial x}\right)+\frac{\partial F}{\partial y}\left(-Ky\frac{\partial F}{\partial y}\right)=0$$

Ecuación X.7: Ecuación del calor en 2D

Se desea resolver el problema en un cuadrilátero. Este es un problema elíptico, en el que hay que imponer condiciones de borde sobre cada lado. Para este problema, pondremos 0 y 3 grados en bordes opuestos, y haremos notar el alumno un principio de las ecuaciones elípticas, que la solución tiene siempre una forma más suave que las condiciones de borde, no pudiendo tener nunca ni un máximo ni un mínimo en la solueión. El modelo de esta ecuación se muestra en el listado X.19.

```
TITLE Steady State Heat equation in 2D

DATA kx := 1.5, ky :=1.5
DOMAIN quad2d := QUADRILATERAL(0,0,5,0,5,5,0,5,ESSENTIAL(EDGE(1,3),3),
                                                ESSENTIAL(EDGE(2,4),0))
MESH m2d := ISOPARAMETRIC(quad2d, QUADRILAT4, ELEMENTS(50,50) )
PDE h2d(0, 0, 0, 1, -kx, 1, -ky, 0, 0, 0, 0, 0, 0, 0, 0, EXPLICIT,
      ITERATIVE(SORCHECKERS) )
m2d.setPDE(h2d)
DYNAMIC
   m2d.STEP()
TIMER delta:=0.1, FINTIM:=0.1
```

Listado X.19: Modelo de la ecuación del calor estática.

Al igual que en casos anteriores, hemos encapsulado todas las primitivas *2D* junto con la ecuación del calor estática, con todos los métodos de resolución iterativos disponibles (ver apartado 5.4.3). El listado X.20 muestra la clase resultante del problema de esta página.

```
CLASS Heat_Q42D_ISO_Q4_CHK
{
* TITLE Elemento cuadrilátero bidimensional para Calor estático
* ABSTRACT Elemento cuadrilátero bidimensional con
** esquina inferior izquierda en (x1,y1) de dimensiones sx x sy
** con una rotación angrot. Discretizado con método
** isoparamétrico (DX x DY), símplice cuadrilátero de 4 nodos.
* AUTHOR Juan de Lara
* EMAIL Juan.Lara@ii.uam.es
NAME name
DATA kx := 1.5, ky :=1.5, DX := 50, DY := 50
DATA x1:=0, y1:=0, sx:=1, sy:=1, angrot:=0
ICON icname := "quadTrp.gif"
DOMAIN rectD := QUADRILATERAL(x1,y1,x1+sx,y1,x1+sx,y1+sy,x1,y1+sy)
rectD.ROTATE(angrot)
MESH rectSol := ISOPARAMETRIC ( rectD, QUADRILAT4, ELEMENTS(DX,DY) )
        PDE h2d(0, 0, 0, 1, -kx, 1, -ky, 0, 0, 0, 0, 0, 0, 0, 0, EXPLICIT,
        ITERATIVE(SORCHECKERS) )
rectSol.setPDE(h2d)
DYNAMIC
  rectSol.STEP()
}
```

Listado X.20: Elemento *Heat_Q42D_ISO_Q4_CHK*.

- Para la séptima página se van a construir dos modelos de la ecuación del calor dependiente del tiempo (esta ecuación se presentó en el problema del calentamiento de las vigas, problema V.6.1, ecuación V.42), un modelo de elementos finitos (que es una simplificación del modelo de las vigas, ya que vamos a resolver la ecuación sobre un cuadrilátero), y un modelo similar a un autómata celular (vamos a permitir valores continuos en los elementos de la matriz). El siguiente listado muestra este último modelo:

```
DATA DX:=20, DY:=20
DATA M[DY;DX], MDF[DY;DX]
DATA M[ROW;COL]:= EXP(-(ROW*0.05+COL*0.05)*(ROW*0.05+COL*0.05)), MDF[ROW;COL]:=0

SETBOUNDARY
  M[ROW;0]    := 0
  M[ROW;DX-1] := 10*SIN(1+ROW*0.05)
  M[0;COL]    := 0
  M[DY-1;COL] := 10*SIN(COL*0.05+1)

DIFFUSSION FILA, COLUMN
  INSW ( FILA, S:=0, S:=M[FILA-1;COLUMN] )
  INSW ( FILA, S+=0, INSW( COLUMN,S+=0,S+=M[FILA-1;COLUMN-1] ) )
  INSW ( FILA,S+=0,INSW(DX-COLUMN-1,S+=0,S+=M[FILA-1;COLUMN+1]))
  INSW ( DY-FILA-1, S+=0, S+=M[FILA+1;COLUMN] )
  INSW ( DY-FILA-1,S+=0,INSW(COLUMN,S+=0,S+=M[FILA+1;COLUMN-1]))
  INSW ( DY-FILA-1, S+=0,INSW( DX-COLUMN-1, S+=0, S+=M[FILA+1;COLUMN+1] ))
  INSW ( COL, S+=0, S+=M[FILA;COLUMN-1] )
  INSW ( DX-COL-1, S+=0, S+=M[FILA;COLUMN+1] )
  DIFFUSSION := S/8
DYNAMIC
  SETBOUNDARY
  MDF[ROW;COL] := DIFFUSSION( ROW, COL )
  M[ROW;COL] := MDF[ROW;COL]
TIMER FINTIM := 2, delta:=0.05
```

Listado X.21: Autómata celular para la difusión del calor.

En este modelo, los nodos de la matriz contendrán un valor, que es la temperatura. En cada iteración, modificamos el valor con una media de la temperatura de sus 8 vecinos (vecindad de *Moore*). Los coeficientes de difusión en x e y se suponen unitarios. Es de hacer notar que este método es bastante parecido a los de diferencias finitas explícitos, sólo que en el caso de un método en diferencias finitas, para esta ecuación sólo se considerarían 4 vecinos (vecindad de *von Neumann*), ya que no hay segundas derivadas cruzadas. Como condición inicial se ha elegido una función exponencial ($e^{-(x+y)(x+y)}$), como se puede observar en la inicialización de la matriz M.

- En la octava página se van a presentar las diversas formas de discretizar dominios que tiene *OOCSMP*, para ello, se va a proceder a discretizar cada una de las primitivas 2D mediante métodos

isoparamétricos y mediante la triangulación de Delaunay (dos *applets* separados). En un tercer *applet* discretizaremos un arco y una pala mezclando ambos métodos de discretización. Los modelos de los dos primeros applets son inmediatos. El modelo del tercer applet se ha diseñado mediante la herramienta *MGEN*, y el código resultante se muestra en el listado X.12.

```
TITLE DISCRETIZING COMPLEX DOMAINS: AN ARC
* MAKE THE ARC
DOMAIN QUAD41:=QUADRILATERAL(-1.000,-1.000,-0.200,-1.000,-0.200, 0.000,-1.000, 0.000)
DOMAIN QUAD42:=QUADRILATERAL( 0.200,-1.000, 1.000,-1.000, 1.000, 0.000, 0.200, 0.0000)
DOMAIN CIRCSECT1:=CIRCSECTOR( 0.000, 0.000, 1.000, 0.200, 0.000, 3.1416)
INTMESH11:=ISOPARAMETRIC(QUAD41,TRIANGLE3L, ELEMENTS(4,4))
INTMESH12:=ISOPARAMETRIC(QUAD42,TRIANGLE3L, ELEMENTS(4,4))
DELMESH11:=DELAUNAY(CIRCSECT1,TRIANGLE3L, ELEMENTS(10,4) , AREA(0.0250) )
INTMESH11.CONCAT(INTMESH12,DELMESH11)
* MAKE THE BLADE
...
TIMER FINTIM:=0
```

Listado X.22: Discretización de un arco y una pala.

- En la página nueve se presenta un ejemplo de mallas móviles. Se ha tomado como base la geomtería del ejemplo de las vigas móviles del problema de la sección VII.3.3.

El resto de las páginas usan modelos que ya han sido descritos en capítulos anteriores.

❑ Paso 3: Adaptar el modelo para cada página

En este paso no es necesario hacer ninguna modificación a los modelos, salvo en el caso de la sexta página, del modelo de la ecuación del calor con elementos finitos y el autómata celular. Para esta página hemos juntado ambos modelos en uno sólo, de forma que se van a ejecutar a la vez. De esta forma vamos a poder comparar el resultado que nos ofrecen ambos métodos. Ambos métodos dan el mismo resultado cuando llegan a un estado estable, pero el modelo de elementos finitos llega antes a la estabilidad.

❑ Paso 4 : Validar los modelos

Para validar los modelos, se ha comprobado el resultado obtenido con los de la literatura, donde ha sido posible.

❑ Paso 5: Decisión de las salidas gráficas

Para el grupo de páginas con solución de ecuaciones sencillas:

- En los primeros modelos (páginas 1,2 y 3) se usará el gráfico tridimensional.
- En la cuarta (transporte no difusivo en dos dimensiones) se usarán dos mapas de isosuperficies, uno para la solución calculada y otra para la solución real.
- En la quinta (ecuación del calor estático en dos dimensiones) se usará un gráfico tridimensional, un mapa de isosuperficies y un listado de los valores de la solución en los nodos.
- En la sexta (calor transitorio con elementos finitos y autómata celular), se usará un mapa de isosuperficies para visualizar la solución de cada método.
- En la séptima (generación de mallas) se usarán gráficos de los nodos de las mallas, que podemos cambiar mediante el uso de la instrucción '\'.

Para el resto de las páginas se discutió la salida gráfica más conveniente en el apartado en que se construyó el modelo.

❑ Paso 6: Inclusión de los elementos multimedia

En este curso se incluirán dos explicaciones dinámicas:

- En el problema del calor usando elementos finitos y el autómata, se presentarán explicaciones textuales con las diferencias entre ambos modelos:

- Una explicación al principio de la simulación, explicando el problema al alumno, indicándole que la difusión se produce de arriba abajo y de derecha a izquierda debido a las condiciones de contorno.
- Otra explicación cuando el modelo de elementos finitos llega al equilibrio, aquí se hace notar al alumno que la difusión del modelo del autómata celular es menor, ya que llega al equilibrio más tarde.
- Otra cuando se llega al tiempo final de la simulación.

- En el modelo de las vigas móviles, se dará una explicación textual de lo que sucede en cada momento, que se colocará debajo del gráfico con los nodos de la malla y del mapa de isosuperficies (casillas *S* y *SE*). Se puede encontrar una imagen de esta página en la figura VIII.3.

❑ Paso 7 : Describir las páginas del curso con *SODA*

Finalmente, hemos de incluir en los modelos las instrucciones necesarias para la generación de código *HTML*, es decir, comentarios, imágenes, etc. En la sección VIII.3 se mostró en detalle la construcción de una de las páginas de teoría y de la página de la piezas móviles.

En la página del autómata celular, se ha incluido una tabla con una gráfica 2D con la condición de contorno y una gráfica 3D con la condición inicial, y el resultado puede verse en la figura X.9.

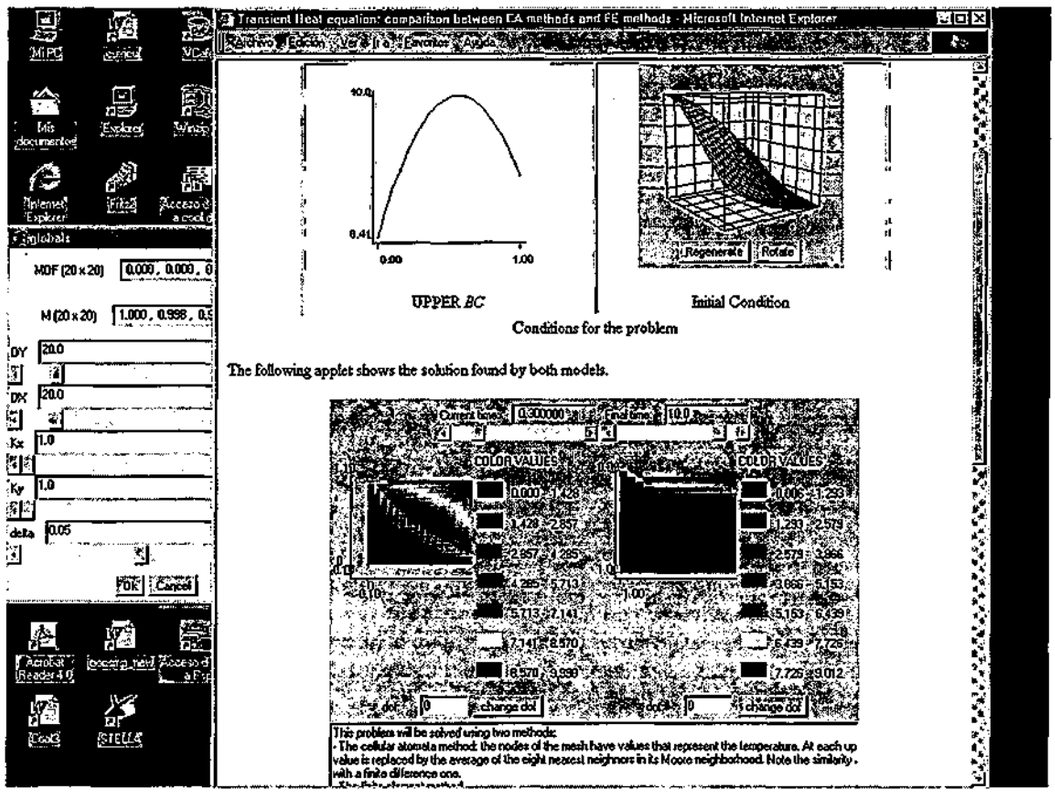


Figura X.9: Página de los modelos del autómata celular y de elementos finitos.

En la figura se puede observar la menor difusión del autómata celular (a la derecha), sin embargo, el resultado cuando ambos llegan al equilibrio es muy similar. El momento en que se tomó la figura, es antes de que el modelo de elementos finitos llegue al equilibrio, el texto explicatorio es por tanto el primero.

XI. Conclusiones, limitaciones y trabajo futuro

En esta sección se presentan las conclusiones del presente trabajo, así como sus limitaciones y las líneas abiertas que se pretende cubrir en un futuro cercano.

XI.1 Conclusiones

Las aportaciones novedosas del presente trabajo han sido:

- Diseño de un nuevo lenguaje de simulación continua, orientado a objetos, y un compilador para el mismo, con las siguientes características:
  - Facilidad de resolución de problemas formados por componentes similares que interactúan.
  - El lenguaje es capaz de tratar eventos discretos. Si se modifica una variable que se integra, se reinicia sólo el integrador asociado a esa variable. Esto es una mejora respecto a [Broe96].
  - Capacidad de combinar la orientación a objetos con la resolución de *PDEs*. Se obtienen más ventajas si se usa la distribución.
  - Se pueden resolver *PDEs* combinando varios métodos de resolución.
  - Posibilidad de diseñar y modificar los dominios, mallas y condiciones de contorno para las *PDEs* en tiempo de ejecución de la simulación, con la herramienta *MGEN*. Esta herramienta puede ser utilizada a través de Internet.
  - Posibilidad de solucionar una *PDE* mezclando varios métodos de solución.
  - Facilidad para generar automáticamente cursos para Internet, ya que *OOCSMP* incorpora una serie de instrucciones de alto nivel (*SODA*) para la descripción de las páginas del curso.
  - Posibilidad de generar código C++, C++/Amulet y Java de las simulaciones. Además es posible la generación automática de una interfaz para ambos casos, en la que el usuario puede explorar el problema (contestar preguntas del tipo *"...qué pasaría si...?"*) mediante el cambio de valores de parámetros.
  - Posibilidad de añadir y borrar objetos durante la ejecución de la simulación, en simulaciones secuenciales. Esto aumenta la posibilidad de experimentación por parte del usuario.
  - Es posible incluir y sincronizar elementos multimedia en las simulaciones, lo que va a permitir realizar las explicaciones en el momento oportuno de la simulación. Esto facilitará la comprensión del modelo al usuario.
  - Posibilidad de generar simulaciones distribuidas con un mínimo cambio del modelo de la simulación secuencial. Se mantiene un único modelo adaptable a ambas situaciones.
  - Los detalles de la tecnología usada para la ejecución de las simulaciones distribuidas es totalmente transparente para el usuario.
  - Es fácil cambiar el esquema de distribución, basta con cambiar el archivo donde se asocian las etiquetas con direcciones de máquinas.
  - El compilador puede generar la documentación del modelo de simulación, usando la información de la tabla de símbolos. Esta documentación se genera en forma de páginas *HTML*.

- El compilador puede generar un modelo de tareas de la interfaz, en caso de que se genere C++/Amulet. Esto hace posible que la simulación generada incluya facilidades en tiempo de ejecución, como ayuda interactiva sobre la interfaz, ejecución de escenarios, etc.

- Se ha construido una librería Java (y C++) para la discretización de dominios y la resolución de *PDEs*.

- Diseño de un procedimiento para la preparación de cursos con componentes de simulación, que incluye la posibilidad de integrar elementos multimedia con la simulación y facilidades para convertir una simulación secuencial en serie.

- Los procedimientos y herramientas desarrolladas en el presente trabajo, han sido utilizados para la generación automática de cuatro cursos para Internet con componentes de simulación. El esfuerzo para generar estos cursos con las herramientas presentadas es tremendamente menor que si se hubieran tenido que construir usando Java y *HTML* directamente.

XI.2 Limitaciones y trabajo futuro

Este trabajo se puede extender en muchas direcciones, debido al campo de aplicación tan amplio en que se encuentra. Se van a exponer algunas lineas abiertas, que coinciden con las limitaciones del presente trabajo.

- En cuanto al lenguaje *OOCSMP*:
  - Se puede mejorar la forma de tratar eventos discretos. Esto podría hacerse creando colas de eventos, y una clase especial *OOCSMP* que fuera el tipo *Evento*. De esta clase se podrían subclasificar todos los eventos discretos que se fueran a tratar en una simulación. Una clase de este tipo podría tener todas las secciones que tiene una clase normal *OOCSMP*. En la sección *DYNAMIC* del evento, se podría incluir el manejo del evento. Además habría que diseñar instrucciones para lanzar eventos, e incluir mecanismos para hacer estadísticas de los eventos, etc.
  - Es posible aumentar las capacidades para la resolución de ecuaciones en derivadas parciales, por ejemplo:
    - Cálculo de deformaciones en estructuras planas. Esto se puede llevar a cabo implementando una nueva primitiva de dominio, que fuera similar a la barra unidimensional, pero que tuviera como coordenadas dos puntos 2D. De esta forma se podrían crear barras que no fueran paralelas a los ejes *X* e *Y*. También habría que diseñar un nuevo generador de mallas, al que se le pudiera indicar todas las barras que componen la estructura. Por último, para la resolución se podrían emplear métodos de elementos finitos que ya existen en la biblioteca que se ha implementado. Este es un campo en el que se está trabajando actualmente.
    - Nuevas primitivas de dominio, por ejemplo, triángulos de lados curvos, o polígonos definidos por un número arbitrario de puntos.
    - Nuevas operaciones sobre mallas, tales como la intersección de mallas, la unión de mallas, el refinamiento, etc.
    - Salidas gráficas para representar convenientemente deformaciones, velocidades y desplazamientos de fluidos (por ejemplo, mediante flechas de distintos tamaños que indiquen la dirección del desplazamiento y su velocidad), etc.
    - Solucionar otro tipo de ecuaciones.
    - Considerar el paso a tres dimensiones espaciales.
    - Posibilidad de importar o exportar geometrías de otros sistemas.
    - Estudiar la posibilidad de cambio automático del método de resolución. Por ejemplo, si la solución varía poco, cambiar a un método menos preciso pero más rápido. Si la solución varía bruscamente en una submalla, cambiar a un método más preciso, pero más costoso computacionalmente.
    - Implementar métodos *multigrid*.
    - Incluir mallas adaptativas.
  - Otra posible línea sería añadir al lenguaje la capacidad de trabajar con números complejos.
  - Puede resultar interesante indicar las unidades de medida en las que están expresadas las distintas variables de la simulación. El compilador se debería encargar de comprobar la coherencia de las unidades de medida de las variables que intervienen en las expresiones.

- Mejora de la forma de reutilización de código *OOCSMP*. Una posible base serían los *"paquetes"* de Java.

- Posibilidad de insertar código nativo Java o C++ dentro de los modelos *OOCSMP*.

- Puede ser interesante poder pasar parámetros por referencia a los bloques *OOCSMP*. Actualmente los vectores, matrices, objetos o colecciones de objetos se pasan por referencia, pero los datos escalares se pasan todos por valor, y no hay forma de expresar que se quieren pasar por referencia.

- Introducir macros en el nivel *OOOCSMP*.

- Herencia múltiple: hasta ahora no se nos ha presentado ningún caso en el que fuese necesario el uso de la herencia múltiple. Además la implementación de este tipo de herencia plantearía dificultades a la hora de traducirla a Java, ya que este lenguaje tampoco la implementa.

- Poder hacer algún tipo de razonamiento cualitativo que permita dar al usuario explicaciones y guías sobre los experimentos que realiza.

- Integración con procedimientos de simulación discreta (autómatas celulares, gramáticas, etc).

- En cuanto al compilador, consideramos que es el primer paso para la creación de una herramienta de autor para la generación de cursos basados en simulación para la web. Esta herramienta de autor debería tener las siguientes características :

  - Debe cubrir todos los pasos de la metodología que hemos desarrollado.

  - Posibilidad de tratar las páginas del curso globalmente. Concepto de curso. Posibilidad de aplicar operaciones a todas las páginas del curso.

  - Diseño gráfico tanto de los modelos como de las páginas del curso.

  - Utilizar *HTML* en los paneles textuales que usamos actualmente como elemento para dar explicaciones dinámicamente.

  - Paneles de realidad virtual. En la actualidad cada objeto *OOCSMP* puede tener asociado un icono, podría ser interesante asociarle un objeto *VRML* [Hart96].

  - La implementación del reproductor de vídeo que hace Java Media Framework, pese a ser multiplataforma, es extremadamente lenta en su inicialización (al menos en la versión utilizada en el presente trabajo). En el futuro se debería cambiar a otras implementaciones de reproductores de vídeo.

  - Mejorar la salida gráfica *CONNECTIONPLOT*, para que al pinchar en un componente (definido como una clase) con el ratón, se pueda ver un diagrama de las ecuaciones que contiene.

  - Posibilidad de ir hacia atrás en el tiempo, cuando el usuario desplace hacia atrás la barra de scroll del control de tiempo actual.

  - Añadir otros elementos multimedia, tales como animaciones.

  - También sería interesante añadir al código generado alguna capacidad de control de las actividades que realiza el alumno en las páginas del curso para la evaluación de las mismas.

  - Idealmente esta herramienta debería estar construida de cara a ser integrada en la web, posiblemente en Java. De esta forma podría aprovechar otros servicios de la web, tal como el modelado distribuido, mediante depósitos de componentes *OOCSMP* distribuidos.

  - Esta herramienta podría integrar en los modelos de simulación elementos que guiaran al usuario en los experimentos, tales como el razonamiento cualitativo, o técnicas de *tutoring* más

sofisticadas que las que hemos llevado a cabo en la sección de Amulet, ya que estas técnicas se basaban en la interfaz, no en el modelo de simulación. Aunque es cierto que la estructura de la interfaz (los botones que se crean, las variables que se pueden modificar) viene determinada por el modelo de simulación.

- Generación por parte del compilador de código Modelica, lo que permitirá exportar los modelos *OOCSMP* a algunas de las herramientas que se han expuesto en el capítulo I. El proceso contrario (leer Modelica y generar *OOCSMP*) también podría ser interesante.

- Se está ultimando una versión definitiva de *C-OOL* para hacerla disponible a través de la web.

- En cuanto a la distribución :

  - Actualmente estamos estudiando la posibilidad de migración de *rmi* a Corba como soporte de las simulaciones distribuidas. Mediante Corba es posible la integración de objetos C++ y Java, de esta forma se podría aprovechar que nuestro compilador es capaz de generar ambos lenguajes.

  - El esquema de distribución que usamos no es óptimo en cuanto a eficiencia, porque no se consideran aspectos como :

    - Enmascarar latencias.
    - Anticipación de datos, caché de datos.
    - Algoritmos más complejos de replicado de computación, en sustitución de comunicaciones.

  - Mecanismos de sincronización globales.

  - Interoperabilidad. Actualmente se pueden incorporar o borrar objetos de la simulación, pero sólo en el caso de que ésta no sea distribuida. Para que esto se pudiera realizar en el caso de simulaciones distribuidas, se necesitarían interfaces comunes en los objetos de simulación, o bien que en tiempo de ejecución un objeto pudiera conocer qué servicios ofrece otro determinado objeto. Esta es una características de los llamados lenguajes reflexivos.

    El concepto de Interoperabilidad nos permitiría, por ejemplo, en el modelo de ecosistemas distribuidos, añadir nuevos ecosistemas en tiempo de ejecución, o nuevas especies. El concepto de interoperabilidad se ha añadido al estándar *HLA*, y está estrechamente relacionado con el de modelado basado en componentes [Page99b].

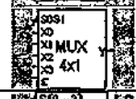
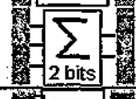
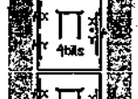
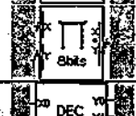
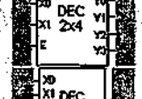
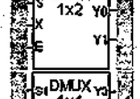
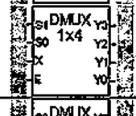
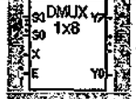
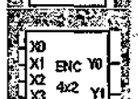
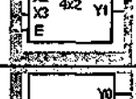
Otras líneas de trabajo que están previstas son:

- Realización de un intérprete de *OOCSMP*, accesible a través de Internet, para añadirlo a *MGEN* e incluirlo en el curso de ecuaciones en derivadas parciales. De esta forma, el alumno será capaz de modificar las ecuaciones durante la simulación. Este trabajo se está realizando en el marco del proyecto *ENCYTEC*, de la *CICYT*.

■ Apéndice A. Biblioteca *OOCSMP* de componentes.

En este apéndice se describe la biblioteca *OOCSMP* de componentes. Esta biblioteca se ha construido en parte aprovechando los modelos de los cursos que se han generado.

Biblioteca de componentes electrónicos :

Componente	Métodos que implementa	Gráfico	Descripción
MUX2x1	DYNAMIC		Multiplexor 2x1.
MUX4x1	DYNAMIC		Multiplexor 4x1.
MUX8x1	DYNAMIC		Multiplexor 8x1.
ADDER1	DYNAMIC		Sumador de 1 bit.
ADDER4	DYNAMIC		Sumador de 4 bits.
ADDER8	DYNAMIC		Sumador de 8 bits.
MULT4x4	DYNAMIC		Multiplicador combinacional de 4 bits.
MULT8x8	DYNAMIC		Multiplicador combinacional de 8 bits.
DEC2X4	DYNAMIC		Decodificador 2x4.
DEC4x16	DYNAMIC		Decodificador 4x16.
DMUX1x2	DYNAMIC		Demultiplexor 1x2.
DMUX1x4	DYNAMIC		Demultiplexor 1x4.
DMUX1x8	DYNAMIC		Demultiplexor 1x8.
ENC4x2	DYNAMIC		Codificador 4x2.
ENC16x4	DYNAMIC		Codificador 16x4.

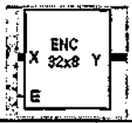
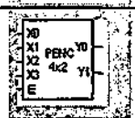
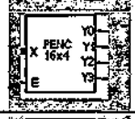
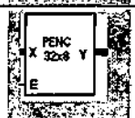
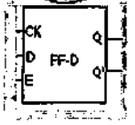
ENC32x8	DYNAMIC		Codificador 32x8.
PENC4x2	DYNAMIC		Codificador con prioridad 4x2
PENC16x4	DYNAMIC		Codificador con prioridad 16x4
PENC32x8	DYNAMIC		Codificador con prioridad 32x8
REG4	LOAD (carga de datos) SHL(desp.lógico izda) SHR(desp.lógico dcha) RHL(rotación izda) RHR(rotación dcha) AHL(desp.aritmético a la izquierda) AHR(desp.aritmético a la derecha)		Registro de 4 bits, con bit de desbordamiento (carry)
REG8	LOAD (carga de datos) SHL(desp.lógico izda) SHR(desp.lógico dcha) RHL(rotación izda) RHR(rotación dcha) ASHL(desp.aritmético a la izquierda) ASHR(desp.aritmético a la derecha)		Registro de 8 bits, con bit de desbordamiento (carry)
REG16	LOAD (carga de datos) SHL(desp.lógico izda) SHR(desp.lógico dcha) RHL(rotación izda) RHR(rotación dcha) ASHL(desp.aritmético a la izquierda) ASHR(desp.aritmético a la derecha)		Registro de 16 bits, con bit de desbordamiento (carry)
COUNT4	ZERO (puesta a cero) DYNAMIC		Contador de 4 bits
COUNT8	ZERO (puesta a cero) DYNAMIC		Contador de 8 bits
COUNT16	ZERO (puesta a cero) DYNAMIC		Contador de 16 bits
FFD	DYNAMIC		Biestable tipo D.

Tabla A.1: Biblioteca de componentes electrónicos.

Actualmente los componentes se están recodificando, cambiando la sección *DYNAMIC* por un método que devuelva un valor, ya que el resultado del bloque se almacenaba en atributos.

Biblioteca de *PDEs*, en una dimensión.

Componente	Métodos que implementa	Descripción
LrNoDiff_B1D_Iso_L2_DF	DYNAMIC	Elemento barra lineal para el transporte no difusivo con el método de DuFort-Frankel.
LrNoDiff_B1D_Iso_L2_EX	DYNAMIC	Elemento barra lineal para el transporte no difusivo con el método de diferencias finitas explícito.
LrNoDiff_B1D_Iso_L2_IM	DYNAMIC	Elemento barra lineal para el transporte no difusivo con el método de diferencias finitas implícito.
LrNoDiff_B1D_Iso_L2_FE	DYNAMIC	Elemento barra lineal para el transporte no difusivo con el método de elementos finitos.
LrNoDiff_B1D_Iso_L3_FE	DYNAMIC	Elemento barra cuadrático para el transporte no difusivo con el método de elementos finitos.
LrDiff_B1D_Iso_L2_DF	DYNAMIC	Elemento barra lineal para el transporte difusivo con el método de DuFort-Frankel.
LrDiff_B1D_Iso_L2_EX	DYNAMIC	Elemento barra lineal para el transporte difusivo con el método de diferencias finitas explícito.
LrDiff_B1D_Iso_L2_IM	DYNAMIC	Elemento barra lineal para el transporte difusivo con el método de diferencias finitas implícito.
LrDiff_B1D_Iso_L2_FE	DYNAMIC	Elemento barra lineal para el transporte difusivo con el método de elementos finitos.
LrDiff_B1D_Iso_L3_FE	DYNAMIC	Elemento barra cuadrático para el transporte difusivo con el método de elementos finitos.
LrHeat_B1D_Iso_L2_DF	DYNAMIC	Elemento barra lineal para la ecuación del calor con el método de DuFort-Frankel.
LrHeat_B1D_Iso_L2_EX	DYNAMIC	Elemento barra lineal para la ecuación del calor con el método de diferencias finitas explícito.
LrHeat_B1D_Iso_L2_IM	DYNAMIC	Elemento barra lineal para la ecuación del calor con el método de diferencias finitas implícito.
LrHeat_B1D_Iso_L2_FE	DYNAMIC	Elemento barra lineal para la ecuación del calor con el método de elementos finitos.
LrHeat_B1D_Iso_L3_FE	DYNAMIC	Elemento barra cuadrático para la ecuación del calor con el método de elementos finitos.

Tabla A.2: Biblioteca de *PDEs* 1-D.

Biblioteca de *PDEs, 2D*, transporte no difusivo :

Componente	Métodos que implementa	Descripción
trNoDiff_Q42D_ISO_Q4_DF	DYNAMIC	Elemento cuadrilátero lineal para el transporte no difusivo con el método de Du Fort-Frankel.
trNoDiff_Q42D_ISO_Q4_EX	DYNAMIC	Elemento cuadrilátero lineal isoparamétrico para el transporte no difusivo con el método de diferencias finitas explícito.
trNoDiff_Q42D_ISO_Q4_IM	DYNAMIC	Elemento cuadrilátero lineal para el transporte no difusivo con el método de diferencias finitas implícito.
trNoDiff_Q42D_ISO_Q4_FE trNoDiff_Q42D_ISO_Q8_FE trNoDiff_Q42D_ISO_T3R_FE trNoDiff_Q42D_ISO_T3L_FE trNoDiff_Q42D_ISO_T6R_FE trNoDiff_Q42D_ISO_T6L_FE	DYNAMIC	Elemento cuadrilátero isoparamétrico (discretizado con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R)) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_Q42D_DEL_Q4_FE trNoDiff_Q42D_DEL_Q8_FE trNoDiff_Q42D_DEL_T3R_FE trNoDiff_Q42D_DEL_T6R_FE	DYNAMIC	Elemento cuadrilátero discretizado mediante delaunay con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3R) y 6 nodos (T6R) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_Q42D_ELL_Q4_FE trNoDiff_Q42D_ELL_Q8_FE trNoDiff_Q42D_ELL_T3R_FE trNoDiff_Q42D_ELL_T3L_FE trNoDiff_Q42D_ELL_T6R_FE trNoDiff_Q42D_ELL_T6L_FE	DYNAMIC	Elemento cuadrilátero discretizado mediante un generador elíptico con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia izquierda (L) o derecha (R) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_Q82D_ISO_Q4_DF	DYNAMIC	Elemento cuadrilátero de ocho nodos lineal para el transporte no difusivo con el método de Du Forte-Frankel.
trNoDiff_Q82D_ISO_Q4_EX	DYNAMIC	Elemento cuadrilátero de ocho nodos lineal isoparamétrico para el transporte no difusivo con el método de diferencias finitas explícito.
trNoDiff_Q82D_ISO_Q4_IM	DYNAMIC	Elemento cuadrilátero de ocho nodos lineal para el transporte no difusivo con el método de diferencias finitas implícito.
trNoDiff_Q82D_ISO_Q4_FE trNoDiff_Q82D_ISO_Q8_FE trNoDiff_Q82D_ISO_T3R_FE trNoDiff_Q82D_ISO_T3L_FE trNoDiff_Q82D_ISO_T6R_FE trNoDiff_Q82D_ISO_T6L_FE	DYNAMIC	Elemento cuadrilátero de ocho nodos isoparamétrico (discretizado con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R)) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_Q82D_DEL_Q4_FE trNoDiff_Q82D_DEL_Q8_FE trNoDiff_Q82D_DEL_T3R_FE trNoDiff_Q82D_DEL_T6R_FE	DYNAMIC	Elemento cuadrilátero de ocho nodos discretizado mediante delaunay con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3R) y 6 nodos (T6R) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_Q82D_ELL_Q4_FE trNoDiff_Q82D_ELL_Q8_FE trNoDiff_Q82D_ELL_T3R_FE trNoDiff_Q82D_ELL_T3L_FE trNoDiff_Q82D_ELL_T6R_FE trNoDiff_Q82D_ELL_T6L_FE	DYNAMIC	Elemento cuadrilátero de ocho nodos discretizado mediante un generador elíptico con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R)) para el transporte no difusivo con el método de elementos finitos.

trNoDiff_T32D_ISO_Q4_DF	DYNAMIC	Elemento triángulo lineal para el transporte no difusivo con el método de Du Fort-Frankel.
trNoDiff_T32D_ISO_Q4_EX	DYNAMIC	Elemento triángulo lineal isoparamétrico para el transporte no difusivo con el método de diferencias finitas explícito.
trNoDiff_T32D_ISO_Q4_IM	DYNAMIC	Elemento triángulo lineal para el transporte no difusivo con el método de diferencias finitas implícito.
trNoDiff_T32D_ISO_Q4_FE trNoDiff_T32D_ISO_Q8_FE trNoDiff_T32D_ISO_T3R_FE trNoDiff_T32D_ISO_T3L_FE trNoDiff_T32D_ISO_T6R_FE trNoDiff_T32D_ISO_T6L_FE	DYNAMIC	Elemento triángulo isoparamétrico discretizado con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_T32D_DEL_Q4_FE trNoDiff_T32D_DEL_Q8_FE trNoDiff_T32D_DEL_T3R_FE trNoDiff_T32D_DEL_T6R_FE	DYNAMIC	Elemento triángulo discretizado mediante delaunay con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3R) y 6 nodos (T6R) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_T32D_ELL_Q4_FE trNoDiff_T32D_ELL_Q8_FE trNoDiff_T32D_ELL_T3R_FE trNoDiff_T32D_ELL_T3L_FE trNoDiff_T32D_ELL_T6R_FE trNoDiff_T32D_ELL_T6L_FE	DYNAMIC	Elemento triángulo de ocho nodos discretizado mediante un generador elíptico con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R)) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_CS2D_ISO_Q4_DF	DYNAMIC	Elemento sector circular lineal para el transporte no difusivo con el método de Du Forte-Frankel.
trNoDiff_CS2D_ISO_Q4_EX	DYNAMIC	Elemento sector circular lineal isoparamétrico para el transporte no difusivo con el método de diferencias finitas explícito.
trNoDiff_CS2D_ISO_Q4_IM	DYNAMIC	Elemento sector circular lineal para el transporte no difusivo con el método de diferencias finitas implícito.
trNoDiff_CS2D_ISO_Q4_FE trNoDiff_CS2D_ISO_Q8_FE trNoDiff_CS2D_ISO_T3R_FE trNoDiff_CS2D_ISO_T3L_FE trNoDiff_CS2D_ISO_T6R_FE trNoDiff_CS2D_ISO_T6L_FE	DYNAMIC	Elemento sector circular isoparamétrico discretizado con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R)) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_CS2D_DEL_Q4_FE trNoDiff_CS2D_DEL_Q8_FE trNoDiff_CS2D_DEL_T3R_FE trNoDiff_CS2D_DEL_T6R_FE	DYNAMIC	Elemento sector circular discretizado mediante delaunay con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3R) y 6 nodos (T6R) para el transporte no difusivo con el método de elementos finitos.
trNoDiff_CS2D_ELL_Q4_FE trNoDiff_CS2D_ELL_Q8_FE trNoDiff_CS2D_ELL_T3R_FE trNoDiff_CS2D_ELL_T3L_FE trNoDiff_CS2D_ELL_T6R_FE trNoDiff_CS2D_ELL_T6L_FE	DYNAMIC	Elemento sector circular discretizado mediante un generador elíptico con cuadriláteros de 4 (Q4) y 8 nodos (Q8), triángulos de 3 (T3) y 6 nodos (T6) alineados hacia la izquierda (L) o la derecha (R)) para el transporte no difusivo con el método de elementos finitos.

Tabla A.2: Biblioteca de *PDEs* 2D para el transporte no difusivo

Existen bibliotecas con componentes similares para el transporte difusivo (*trDiff_xxx*), calor transitorio (*trHeat_xxx*) y calor estático (*Heat_xxx*).

■ Apéndice B. Gramática de *OOCSMP*.

A continuación se describe el lenguaje *OOCSMP* en notación *BNF*. Está compuesta por 330 reglas. Los tokens aparecen en negrita. La siguiente lista indica dónde se define cada uno de los elementos de *OOCSMP*:


```
[1]   <model>              ::= <declarat-section> <blocks-dcl>

[2]   <declarat-section>   ::= <include-dcl> <declarat-section>      |
[3]                            <title-dcl> <declarat-section>       |
[4]                            <data-dcl> <declarat-section>        |
[5]                            <class-dcl> <declarat-section>       |
[6]                            <Domain-dcl> <declarat-section>      |
[7]                            <Mesh-dcl> <declarat-section>        |
[8]                            <PDE-dcl> <declarat-section>

[9]   <Data-dcl>           ::= <DATA> <identifier> <rest-id-dcl>          |
[10]                           <identifier> <object-dcl>

[11]  <rest-id-dcl>        ::= <[> <pos-integer> <]> <id-init> <rest-list-ids>    |
[12]                           <[> <pos-integer> <;> <pos-integer> <]>
                               <id-init> <rest-list-ids>                          |
[13]                           <id-init> <rest-list-ids>                          |
[14]                           <identifier> <object-dcl>

[15]  <id-init>            ::= <:=> <Expresion>                           |
[16]                           <:=> <number> <rest-numbers>
[17]                           λ

[18]  <rest-list-ids>      ::= <,> <identifier> <rest-id-dcl>|
                               λ

[19]  <pos-integer>        ::= <ciphre> <rest-integer>

[20]  <rest-integer>       ::= <ciphre> <pos-integer>|
[21]                           λ

[22]  <object-dcl>         ::= <identifier> <obj-or-collection>

[23]  <obj-or-collection>  ::= <(> <parameters>                   |
[24]                           <:=> <identifier> <rest-objects>

[25]  <rest-objects>       ::= <,> <identifier>                           |
[26]                           λ

[27]  <parameters>         ::= <Expresion> <rest-parameters> <)>    |
```

```
[28]                              <"> <text> <">                          |
[29]                              <)>

[30]   <rest-parameters>   ::=  <,> <Expresion> <rest-parameters>     |
[31]                              λ

[32]   <class-dcl>         ::=  <CLASS> <identifier> <rest-class-dcl>

[33]   <rest-class-dcl>    ::=  <;>                                          |
[34]                              <:> <identifier> <{> <class-model> <}>      |
[35]                              <{> <class-model> <}>

[36]   <class-model>       ::=  <datain-dcl> <more-datain-dcl> <blocks-cl-dcl>      |
[37]                              <blocks-cl-dcl>

[38]   <datain-dcl>        ::=  <data-dcl>     |
[39]                              <icon-dcl>     |
[40]                              <name-dcl>

[41]   <more-datain-dcl>   ::=  <datain-dcl> <more-datain-dcl>       |
[42]                              λ

[43]   <blocks-cl-dcl>     ::=  <initial-block> <usr-blocks-dcl> <dynamic-cl-dcl>
                                  <ctrl-data-cl-dcl>

[44]   <ctrl-data-cl-dcl> ::=  <finish-dcl> <Output-dcl> |
[45]                              <Output-dcl> <finish-dcl>

[46] <Domain-decl>         ::=  <DOMAIN> <identifier> <:=> <Quadrilateral-decl>   |
[47]                              <DOMAIN> <identifier> <:=> <CircSector-decl>      |
[48]                              <DOMAIN> <identifier> <:=> <Triangle-decl>        |
[49]                              <DOMAIN> <identifier> <:=> <Quad8-decl>           |
[50]                              <DOMAIN> <identifier> <:=> <Bar-decl>             |
[51]                              <DOMAIN> <identifier> <:=> <MGEN> <(> <dynamicfuncs>

[52] <Bar-decl>            ::=  <BAR> <(>
                                  <Expresion> <,> <Expresion> <Conditions-decl> <)>

[53] <Quadrilateral-decl>::=  <QUADRILATERAL> <(>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><Conditions-decl> <)>

[54] <Triangle-decl>       ::=  <TRIANGLE><(>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><Conditions-decl> <)>

[55] <Quad8-decl>          ::=  <QUAD8><(>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><,>
                                  <Expresion><,><Expresion><Conditions-decl> <)>

[56] <CircSector-decl> ::=  <CIRCSETCTOR><(>
                                  <Expresion><,><Expresion><,>
                                  <rad1><,><rad2><,>
                                  <initAng><,><finAng> <Conditions-decl> <)>

[57] <dynamicfuncs>        ::=  <DYNAMIC><(><Expresion><)><moreDynamics> |
[58]                              <)>

[59] <moreDynamics>        ::=  <,><dynamicfuncs>       |
```

```
[60]                                  <)>

[61] <Conditions-decl> ::=    λ       |
[62]                          <Condition-decl><Conditions-decl>

[63] <Condition-decl>  ::=    <,><INITIAL><(><whereInitial>                          |
[64]                          <,><INITIALDT><(><whereInitial>                        |
[65]                          <,><ESSENTIAL><(><whereBoundary><,><Expresion><)>      |
[66]                          <,><NATURAL><(><whereBoundary><,><Expresion><)>       |
[67]                          <,><PERIODIC><(><whereBoundary><)>

[68] <whereInitial>    ::=    <DOF> <(> <Expresion> <)> <,> <Expresion> <)> |
[69]                          <Expresion> <)>

[70] <whereBoundary>   ::=    <DOF> <(> <Expresion> <)> <,> <whereBC> |
[71]                          <whereBC>

[72] <whereBC>         ::=    <EDGE>      <(>    <where>  <)>   |
[73]                          <CORNER>    <(>    <where>  <)>   |
[74]                          <NODE>      <(>    <where>  <)>

[75] <where>           ::=    <expresion><restoWhere>

[76] <restoWhere>      ::=    <:><expresion><restoLista>    |
[77]                          <restoLista>                  |
[78]                          <)>

[79] <restoLista>      ::=    <,> <where>

[80] <Mesh-decl>       ::=    <MESH> <identifier> <:=> <TypeMesh-decl>

[81] <TypeMesh-decl>   ::=    <ISOPARAMETRIC> <Iso-decl>        |
[82]                          <DELAUNAY> <Del-decl>             |
[83]                          <ELLIPTIC> <Ell-decl>             |
[84]                          <MGEN> <(> <domain-const>

[85] <domain-const>    ::=    <domain-id> <)> |
[86]                          <)>

[87] <Iso-decl>        ::=    <(><domain-name> <,> <simplex-name> <,>
                              <ELEMENTS> <(> <Expresion> <Resto2d> <smooth>

[88] <simplex-name>    ::=    <QUADRILAT4> |
[89]                          <TRIANGLE3L> |
[90]                          <TRIANGLE3R> |
[91]                          <TRIANGLE6L> |
[92]                          <TRIANGLE6R> |
[93]                          <LINE2>      |
[94]                          <LINE3>

[95] <Resto2d>         ::=    <,><Expresion><)>      |
[96]                          <)>

[97] <smooth>          ::=    <,><SMOOTH> <)>       |
[98]                          <)>

[99] <Del-decl>        ::=    <(> <domain-name> <,> <simplex-name> <,>
                              <ELEMENTS> <(> <Expresion> <Resto2d> <constraints>

[100] <constraints>    ::=    <,> <constraint> |
[101]                         <)>

[102] <constraint>     ::=    <AREA> <(> <Expresion> <)> <constraints>     |
                              <ANGLE> <(> <Expresion> <)> <constraints>    |
                              <SIZE> <(> <Expresion> <)> <constraints>     |
                              <SMOOTH> <)>

[103] <Ell-decl>       ::=    <(> <domain-name> <,> <simplex-name> <,>
```

```
                        <ELEMENTS> <(> <Expresion> <Resto2d> <PDE-Sys-id>

[104] <PDE-decl>    ::= <PDE> <identifier> <(> <System-dcl> <Funcs1> <PDE-METHOD>

[105] <Functions>   ::= <Funcs1> <)>   |
[106]                   <)>

[107] <Funcs1>      ::= <Expresion> <,> <Expresion> <,> <Expresion> <,>
                        <Expresion> <,> <Expresion> <,> <Expresion> <,>
                        <Expresion> <,> <Expresion> <,> <Expresion> <,>
                        <Expresion> <,> <Expresion> <,> <Expresion> <,>
                        <Expresion> <,> <Expresion> <,> <Expresion>

[108] <System>      ::= λ |
[109]                   System <(> ( <pde-id><,> )* <pde-id> <)> <,>

[110] <PDE-METHOD>  ::= <FEM> <lumping>      |
[111]                   <EXPLICIT> <Schemes> |
[112]                   <IMPLICIT> <Schemes> |
[113]                   <DUFORT> <Schemes>

[114] <Schemes>     ::= <,> <TScheme> <(> <Scheme> <)> < SpSchemes> |
[115]                   <)>

[116] <SpSchemes>   ::= <,> <XScheme> <(> <Scheme> <)> < Yscheme>   |
[117]                   <)>

[118] <Yscheme>     ::= <,> <YScheme> <(> <Scheme> <)> <elliptic-meth> |
[119]                   <)>

[120] <elliptic-meth> ::= <ITERATIVE> <)> <it-method> <)>     |
[121]                   <)>

[122] <Scheme>      ::= <CENTRAL>    |
[123]                   <FORWARD>    |
[124]                   <BACKWARDS>

[125] <lumping>     ::= <)> |
[126]                   <LUMPING> <(> <Lump> <)> < NewMark-Params>

[127] <Lump>        ::= <NOLUMPING>          |
[128]                   <ROWSUMLUMP>         |
[129]                   <PROPLUMP>

[130] <NewMark-Params> ::= <)>               |
[131]                   <,> <ALPHA> <(> <Expresion> <)> <Gamma>

[132] <Gamma>       ::= <)> |
[133]                   <,> <GAMMA> <(> <Expresion> <)> <)>

[134] <Output-decl> ::= <PLOT> <location> <legend> <scale-panel> <machine>
                            <scale> <vars-plot>  |
[135]                   <PLOT2D> <location> <machine> <vars-plot>                |
[136]                   <PLOT3D> <location> <machine> <scale> <vars-plot>        |
[137]                   <ISOPLOT> <location> <machine> <scale> <vars-plot>       |
[138]                   <GRIDPLOT> <location> <machine> <vars-plot>              |
[139]                   <ICONICPLOT> <location> <machine> <vars-iconicplot>      |
[140]                   <CONNECTIONPLOT> <location> <machine> <method-form>      |
[141]                   <PRINT> <location> <machine> <vars-plot>                 |
[142]                   <IMAGEPANEL> <location> <multimedia-conds>       |
[143]                   <TEXTPANEL> <location> <multimedia-conds>        |
[144]                   <VIDEOPANEL> <location> <multimedia-conds>       |
[145]                   <AUDIOPANEL> <location> <multimedia-conds>       |
[146]                   λ

[147] <location>    ::= <[> <position> <restPosition> |
[148]                   λ
```

```
[149] <restPosition>     ::=   <,> <position> <restPosition> |
[150]                          <]>

[151] <position>         ::=   <N>    |
[152]                          <C>    |
[153]                          <S>    |
[154]                          <E>    |
[155]                          <W>    |
[156]                          <NE>   |
[157]                          <NW>   |
[158]                          <SE>   |
[159]                          <SW>   |
[160]                          <WINDOW>

[161]  <legend>          ::=   <[> <LG> <=> <position><restPosition> <,>     |
[162]                          λ

[163]  <scale-panel>     ::=   <[> <SC> <=> <position><restPosition> <,>     |
[164]                          λ

[165]  <machine>         ::=   <[> <MACHINE> <=> <identifier> <]> <,> |
[166]                          λ

[167]  <scale>           ::=   <real> <,> <real> <rest-scale>|
[168]                          λ

[169]  <rest-scale>      ::=   <,> <real> <,> <real>                  |
[170]                          λ

[171]  <vars-plot>       ::=   <identifier> <rest-vars>

[172]  <rest-vars>       ::=   <,> <vars-plot> |
[173]                          λ

[174]  <vars-iconicplot>::=    <identifier> <filename-opt>

[175]  <filename-opt>      ::= <"> <filename-gif> <"> <rest-iconicplot>     |
[176]                          <,> <vars-iconicplot>                        |
[177]                          λ

[178]  <rest-iconicplot>   ::= <,> <vars-iconicplot>          |
[179]                          λ

[180]  <filename-gif>      ::= <identifier> <.> <GIF>

[181]  <method-form>       ::= <identifier> <.> <identifier> <,> <LED7>     |
[182]                          <identifier> <,> <LED7>                      |
[183]                          <LED7>                                       |
[184]                          λ

[185]  <multimedia-conds>  ::= <START> <(> <Bool-Expresion> <)> <,> <"> <file> <">
                               <more-conditions>

[186]  <more-conditions>   ::= <,> <multimedia-conds>              |
[187]                          <,> <DEFAULT> <,> <"> <file> <">    |
[188]                          λ

[189]  <Title-dcl>         ::= <TITLE> <text>

[190]  <text>              ::= (<letter>)*

[191]  <timer-dcl>         ::= <TIMER> <ctrl-dcl>

[192]  <ctrl-dcl>          ::= <delta> <:=> <real-value> <rest-ctrl-dcl>    |
[193]                          <FINTIM> <:=> <real-value> <rest-ctrl-dcl>   |
[194]                          <PLdelta> <:=> <real-value> <rest-ctrl-dcl>  |
[195]                          <PRdelta> <:=> <real-value> <rest-ctrl-dcl>  |
```

```
[196]                          <MAXdelta> <:=> <real-value> <rest-ctrl-dcl>|
[197]                          <MINdelta> <:=> <real- value> <rest-ctrl-dcl>

[198]  <rest-ctrl-dcl>    ::= <,> <ctrl-dcl> |
[199]                         λ

[200]  <finish-dcl>       ::= <FINISH> <Bool-Expresion>

[201]  <method-dcl>       ::= <METHOD> <machine> <ode-method> <rest-methods>   |
[202]                         λ

[203]  <ode-method>       ::= <ADAMS>              |
[204]                         <RKS>                |
[205]                         <RKSFX>              |
[206]                         <SIMP>               |
[207]                         <RECT>               |
[208]                         <TRAPZ>

[209]  <rest-methods>     ::= <,><[><MACHINE><=><identifier><]>  <ode-method>
                                                                 <rest-methods> |
[210]                         λ

[211]  <icon-dcl>         ::= <ICON> <identifier> <rest-icon-dcl>

[212]  <rest-icon-dcl>    ::= <=> <"> <filename-gif> <">    |
[213]                         λ

[214]  <name-dcl>         ::= <NAME> <identifier>

[215]  <include-dcl>      ::= <INCLUDE> <"> <text> <">

[216]  <blocks-dcl>       ::= <initial-block> <instructions> <final-instruc>

[217]  <final-instruc>    ::= <finish-dcl> <final-instruc>         |
[218]                         <timer-dcl> <final-instruc>          |
[219]                         <method-dcl> <final-instruc>         |
[220]                         <Output-dcl> <final-instruc>         |
[221]                         <alter-simul> <final-instruc>

[222]  <alter-simul>      ::= <\> <rest-alter>

[223]  <rest-alter-simul> ::= <text> <alter-instruc>               |
[224]                         <alter-instruc>

[225]  <alter-instruc>    ::= <timer-dcl> <alter-instruc>          |
[226]                         <data-dcl> <alter-instruc>           |
[227]                         <Title-dcl> <alter-instruc>          |
[228]                         <Output-dcl> <alter-instruc>         |
[229]                         <assignment> <alter-instruc>

[230]  <instructions>     ::= <class-model> <instructions>         |
[231]                         <usr-blocks-dcl> <instructions>      |
[232]                         <DYNAMIC> <instructions>             |
[233]                         <instruc> <instructions>             |
[234]                         λ

[235]  <usr-blocks-dcl>   ::= <identifier> <parameter-dcl>

[236]  <assign-meth-evt>  ::= <instruc> <assign-meth-evt>          |
[237]                         λ

[238]  <parameter-dcl>    ::= <one-par-dcl> <more-parameter-dcl>   |
[239]                         λ

[240]  <instruc>          ::= <assignment>   |
[241]                         <methd-invoc>  |
[242]                         <evt-catch>

[243]  <methd-invoc>      ::= <object-dcl>
```

```
[244] <evt-catch>          ::= <INSW> <(> <Expresion> <,>   <instruc-void> <,>
                                                           <instruc-void> <)>   |
[245]                          <FCNSW><(> <Expresion> <,>  <instruc-void> <,>
                                                           <instruc-void> <,>
                                                           <instruc-void> <)>

[246] <more-parameter-dcl>::= <,> <one-par-dcl>      |
[247]                          λ

[248] <instruc-void>       ::= <instruc>      |
[249]                          λ

[250] <one-par-dcl>        ::= <identifier>                          |
[251]                          <identifier> <identifier>             |
[252]                          <identifier> <identifier> <[> <]>     |
[253]                          <identifier> <[> <]>                  |
[254]                          <identifier> <[> <;> <]>

[255] <initial-block>      ::= <INITIAL>      |
[256]                          λ

[257] <dynamic-cl-dcl>     ::= <DYNAMIC> <parameter-dcl> <assign-meth-evt>      |
[258]                          λ

[259] <assignment>         ::= <array-id> <:=> <Expresion>   |
[260]                          <array-id> <+=> <Expresion>   |
[261]                          <array-id> <-=> <Expresion>   |
[262]                          <array-id> <*=> <Expresion>   |
[263]                          <array-id> </=> <Expresion>

[264] <assignments>        ::= <assignment> <assignments>    |
[265]                          <assignment>

[266] <array-id>           ::= <identifier> <[> <Expresion> <]>                    |
[267]                          <identifier> <[> <Expresion> <;> <Expresion> <]>    |
[268]                          <identifier>

[269] <Expresion>          ::= <identifier> <(> <parameters> <compar> <mult> <add-substr>|
[270]                          <identifier> <.> <identifier> <(>
                                      <parameters> <compar> <mult> <add-substr>   |
[271]                          <array-id> <compar> <mult> < add-substr >           |
[272]                          <(> <Expresion> <)> <compar> <mult> <add-substr>    |
[273]                          <number> <compar> <mult> <add-substr>               |
[274]                          <+> <factor> <compar> <mult> <add-substr>           |
[275]                          <-> <factor> <compar> <mult> <add-substr>

[276] <add-substr>         ::= <+> <Expresion>      |
[277]                          <-> <Expresion>      |
[278]                          λ

[279] <compar>             ::= <&&> <factor> <mult> <compar>       |
[280]                          <||> <factor> <mult> <compar>       |
[281]                          <=> <factor> <mult> <compar>        |
[282]                          <#> <factor> <mult> <compar>        |
[283]                          <<> <factor> <mult> <compar>        |
[284]                          <>> <factor> <mult> <compar>        |
[285]                          <<=> <factor> <mult> <compar>       |
[286]                          <>=> <factor> <mult> <compar>

[287] <mult>               ::= <*> <factor> <mult>          |
[288]                          </> <factor> <mult>          |
[289]                          <%> <factor> <mult>          |
[290]                          <**> <factor> <mult>

[291] <factor>             ::= <identifier> <(> <parameters>       |
[292]                          <array-id>                          |
[293]                          <(> <Expresion> <)>                 |
```

```
[294]                          <number>                                          |
[295]                          <+> <factor>                                      |
[296]                          <-> <factor>

[297] <SODA-instr>      ::= <DESCRIPTION> <text>                                 |
[298]                       <TRANSLATE> <"> <macroProto> <"> <,> <"> <text> <">  |
[299]                       <STYLE> <"> <styleId> <"> <,> <"> <style-dcl> <">    |
[300]                       <LINK> <SODA-pos> <"> <HTML-page> <"><,><"> <text> <"> |
[301]                       <IMAGE> <SODA-pos> <"> <file><"><SODA-pos><"> <text> <"> |
[302]                       <BAR> <pos-percent>                                  |
[303]                       <TABLE> <rest-table>                                 |
[304]                       <MODEL> <numFilCols><,><SODA-pos><"><file><"><codebase> |
[305]                       <3DGRAPHIC> <rest3DGraphic>                          |
[306]                       <ISOGRAPHIC> <rest3DGraphic>                         |
[307]                       <2DGRAPHIC> <rest2DGraphic>

[308] <pos-percent>     ::= <[> <position> <rest-percent>                        |
[309]                       λ

[310] <rest-percent>    ::= <,> <number>                                         |
[311]                       <]>

[312] <rest2DGraphic>   ::= <SODA-pos> <numFilCols> <,>
                            <[> <number> <]> <,> <[> <real> <,> <real> <]> <,>
                            <Expresion> <,> <"> <text> <"> <,> <"> <text> <">

[313] <rest3DGraphic>   ::= <SODA-pos> <numFilCols> <,> <numFilCols> <,>
                            <[> <real> <,> <real> <,> <real> <,> <real> <]> <,>
                            <Expresion> <,> <"> <text> <"> <,> <"> <text> <">

[314] <macroProto>      ::= <identifier> <macroProto>     |
[315]                       λ

[316] <styleId>         ::= <\> <identifier> <key-void>

[317] <key-void>        ::= <{>     |
[318]                       λ

[319] <style-dcl>       ::= <styleId> <style-dcl> |
[320]                       λ

[321] <rest-table>      ::= <"> <text> <"> <,> <numFilCols> <cells> |
[322]                       <numFilCols> <cells>

[323] <cells>           ::= <SODA-pos> <rest-cells>       |
[324]                       <rest-cells>

[325] <rest-cells>      ::= <"> <text> <"> <more-cells>

[326] <more-cells>      ::= <"> <text> <"> <,> <more-cells>       |
[327]                       λ

[328] <numFilsCols>     ::= <[> <number> <;> <number> <]>

[329] <SODA-pos>        ::= <[> <position> <]>

[330] <number>          ::= ( <digit> )+
```

Apéndice C. Las bibliotecas de resolución de *PDEs*.

En este apéndice, se presentan algunos aspectos interesantes del diseño de las librerías de cálculo numérico Java y C++ que se han desarrollado. Las bibliotecas Java ocupan aproximadamente 2Mb de código fuente, mientras que las bibliotecas de resolución de PDEs escritas en C++ ocupan 1Mb de código fuente.

Las bibliotecas para la resolución de *PDEs* desarrolladas en C++ y en Java tienen la misma arquitectura. Han sido desarrolladas usando programación orientada a objetos, cuidando principalmente la extensibilidad. Se ha intentado que sea fácil añadir nuevos dominios básicos, símplices, algoritmos de resolución o generadores de malla. Para ello se han distribuido las clases en cinco jerarquías, de acuerdo a su función:

- Tratamiento de puntos geométricos.
- Símplices.
- Dominios.
- Mallas.
- Algoritmos de resolución de *PDEs*.

A continuación se describe brevemente cada uno de estos grupos.

C.1 Clases para el tratamiento de puntos geométricos.

La clase principal de esta jerarquía es la clase *pointInfo*, que recoge los atributos básicos de un punto (coordenadas x e y, tipo de condición de contorno si la hay, etc.). Como es frecuente tener colecciones de puntos (por ejemplo en dominios, mallas, etc.), se ha optado por separar todos los métodos de tratamiento de puntos en una clase separada, estática, llamada *pointHandler*, de esta forma, el ahorro de memoria es considerable, debido a que los métodos no se replican en cada objeto, sino que sólo están en la clase *pointHandler*.

El algoritmo de triangulación necesita tener información además sobre lados y sobre triángulos. Para ello, se ha creado una clase llamada *edge*, que básicamente consta de dos objetos *pointInfo* y dos objetos *triangle* (para mayor eficiencia, los triángulos izquierdo y derecho que tiene cada lado), y de una clase *triangle*, con un vector de tres objetos *pointInfo* y otro vector de tres objetos *edge*. Al igual que la clase *pointHandler*, existen dos clases estáticas: *edgeHandler* y *triangleHandler*.

Así mismo se han implementado listas de puntos y lados (que heredan de la clase estándar Java *Vector*) con operaciones especiales de búsqueda, borrado y reemplazamiento.

Esta jerarquía se ha incluido en Java en un paquete llamado *csmp.pde.points*.

C.2 Clases que implementan símplices

Cada tipo de símplice está encapsulado en una clase, que hereda de una clase base llamada *Simplice*. Esta clase declara como abstractas las funciones de forma, y las derivadas respecto a x e y. Además implementa funciones para calcular el *Jacobiano* del elemento y su inverso, las funciones $x(\xi, \eta)$ e $y(\xi, \eta)$ y sus derivadas respecto a ξ y η.

Cada una de las clases que implementan un símplice (*line2* y *line3* para una dimensión, *Quad4*, *Quad8*, *tri3* y *tri6* para dos dimensiones) implementan las funciones de forma y sus derivadas, heredan el resto de los métodos.

Esta jerarquía se ha incluido en Java en un paquete llamado *csmp.pde.simplex*.

C.3 Clases que implementan dominios

Cada dominio se ha implementado como una clase. Todas ellas heredan de la clase *domain*. Esta clase contiene información sobre las condiciones iniciales y de contorno, y el número de puntos y ecuaciones (grados de libertad) del dominio. También declara como abstractos métodos para calcular $x(\xi, \eta)$ e $y(\xi, \eta), \xi(x, y)$ y $\eta(x, y)$ y sus derivadas primeras y segundas respecto a x e y, y para las

transformaciones geométricas. Implementa un método que una cadena de caracteres con la sintaxis del dominio en *OOCSMP* (útil si se usa la clase desde *MGEN*).

Las clases que implementan dominios (*lineDomain* para una dimensión, *quadDomain*, *q8Domain*, *triangleDomain* y *circSectDomain* para dos dimensiones) contienen atributos para guardar los puntos que definen el dominio (y los ángulos en el caso del sector circular) e implementan los métodos para calcular $x(\xi, \eta)$ e $y(\xi, \eta)$,$\xi(x, y)$ y $\eta(x, y)$ y sus derivadas primeras y segundas respecto a x e y, y para implementar las operaciones geométricas.

Las condiciones de contorno e iniciales se implementan como objetos que deben implementar la interfaz *funcptr*. En esta interfaz se han declarado los prototipos de las 15 funciones de la ecuación modelo.

Esta jerarquía se ha incluido en Java en un paquete llamado *csmp.pde.domains*.

■ C.4 Clases que implementan mallas

Cada tipo de generador de malla, se ha implementado en una clase, pero todas heredan de la clase *mesh*. Esta clase representa la discretización de un dominio básico, y tiene información sobre el símplice usado, el dominio que discretiza, el algoritmo de resolución que se le ha asociado, información sobre los puntos que contiene el contorno, una lista con los puntos que ha generado la malla, tanto para el dominio computacional como para el físico, las mallas con las que se ha concatenado, etc.

Implementa métodos para rellenar los puntos del dominio computacional, identificar los puntos del contorno, concatenar una malla, separar una malla, rellenar los puntos con condiciones de contorno, realizar las operaciones geométricas, devolver la coordenada X o Y más a la izquierda, más a la derecha, más arriba o más abajo, etc.

Declara como abstractos métodos para devolver una cadena con la sintaxis *OOCSMP*, para calcular una matriz de conectividad (usada por el método de los elementos finitos) y para suavizar la malla.

Esta biblioteca implementa tres generadores de malla:

- *interpMesh*, que calcula los puntos del dominio físico usando los puntos del dominio computacional (que calcula la clase *mesh*) y las funciones $x(\xi, \eta)$ e $y(\xi, \eta)$. Implementa el método que realiza el suavizado, el que calcula la matriz de conectividad y el que devuelve la sintaxis.
- *delaunayMesh*, que realiza una triangulación incremental, para lo cual usa una lista de triángulos 'buenos' (que cumplen las restricciones) y 'malos' (que no las cumplen) y una lista de lados. El algoritmo no usa los puntos del dominio computacional, y sigue el algoritmo que se puede encontrar en [Hsua97]. Implementa el método que realiza el suavizado, el que calcula la matriz de conectividad y el que devuelve la sintaxis, y tiene un método para dividir cada triángulo generado en tres cuadriláteros, en caso necesario.
- *EllipticMesh*, que contiene una malla para resolver las *PDEs* que darán lugar a las coordenadas y las *PDEs* que se van a resolver. Implementa el método que realiza el suavizado, el que calcula la matriz de conectividad y el que devuelve la sintaxis.

En Java, todas estas clases se han incluido en un paquete llamado *csmp.pde.meshes*.

■ C.5 Clases que implementan los algoritmos de resolución.

Todos los '*solvers*' heredan de una clase llamada *PDE_Solver*. Esta clase contiene información sobre las 15 funciones de la ecuación modelo y la malla a la que está asociado. Declara métodos abstractos para las transformaciones geométricas, para concatenarse o separarse de otra malla, para resolver dependiendo de si la malla es 1D, si es una ecuación 2D estática o con análisis transitorio.

Los algoritmos de diferencias finitas, heredan además de una clase llamada *FD_Solver* (que a su vez hereda de *PDE_Solver*). La jerarquía de clases puede observarse en la figura C.1.

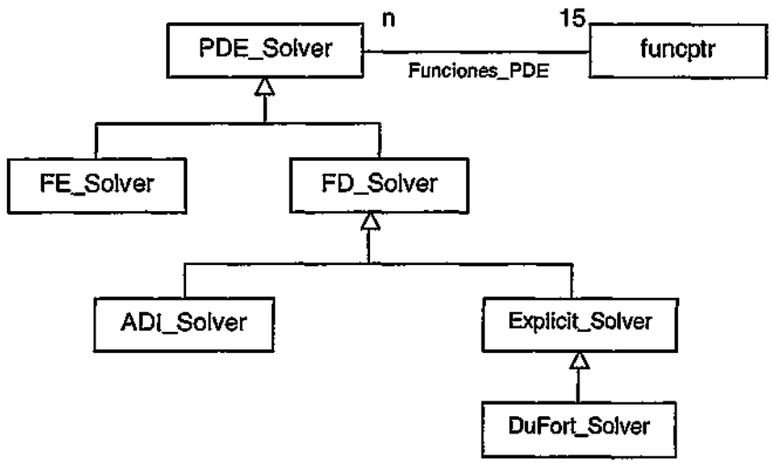


Figura C.1: Diagrama de las clases que implementan los algoritmos de solución de *PDEs*

- La clase *FD_Solver* define las matrices donde se solucionará la ecuación, implementa los algoritmos de solución en los bordes, un esquema general de discretización, métodos que rellenan las matrices con las condiciones del problema, métodos para la resolución iterativa de problemas elípticos, etc.

- La clase *Explicit_Solver* implementa los algoritmos completos de solución, usando la discretización definida en la clase *FD_Solver*.

- La clase *DuFort_Solver* sobreescribe algunas de las discretizaciones de las derivadas que hacía la clase *FD_Solver*, los algoritmos de solución los hereda de *Explicit_Solver*.

- La clase *ADI_Solver* implementa los algoritmos para la solución mediante *ADI* o Crank-Nicolson.

- La clase *FE_Solver* implementa los algoritmos para la solución mediante elementos finitos.

Las clases *FE_Solver*, *ADI_Solver*, y *Explicit_Solver* tienen en cuenta si el *solver* asociado a alguna malla concatenada con su malla asignada tiene el mismo método de resolución, si es así, el *solver* ha de configurar el tamaño de sus vectores y matrices internos, de forma que todas las mallas contiguas con igual método se ejecutan como si fuesen una sola malla, y las ejecuta el primer *solver*. Estas clases implementan los métodos *concatenate* y *detach*, que comprueban si la malla a concatenar o a separar resuelve con igual método la ecuación, y si es así, modifica el tamaño de vectores, copia la solución u otros valores de un *solver* a otro, etc.

Glosario

- Avi: Audio Video Interleave. Formato de vídeo diseñado por Microsoft, muy usado en entornos Windows. Requiere un hardware especial para su captura y compresión, pero no para la descompresión y reproducción [Bust94].

- Applet: Es un programa Java que puede ser llamado desde una página *HTML* y se ejecuta en la máquina del cliente.

- BNF: Backus Normal Form, una notación para describir gramáticas formalmente.

- DHTML: Dynamic *HTML*. Es un conjunto de nuevas tecnologías que extienden el lenguaje *HTML*, y añaden funcionalidades a las páginas Web, como animaciones e 'inteligencia' en el lado cliente que antes necesitaban Applets , ActiveX o similares.

- Framework: Grupos de clases que colaboran entre sí, y que pueden aislarse de las restantes clases gracias al encapsulamiento [Sier99].

- Gif : Es un formato de imagen, creado por CompuServe como un formato de imagen totalmente independiente del ordenador. Es el más usado para almacenar imágenes de 8 bits digitalizadas o escaneadas [Bust94]. Usado en documentos *HTML*.

- Hilo o thread: También conocidos como miniprocesos, o procesos ligeros [Tane93]. Cada hilo se ejecuta de forma secuencial y tiene su propia pila, su contador de programa, etc. En muchos sistemas distribuidos, es posible tener varios hilos. Si el sistema no es multiprocesador, los hilos comparten CPU, y se ejecutan en tiempo compartido.

- HTML : HyperText Markup Language. Es el lenguaje en que están escritos los documentos que podemos visualizar con un navegador de Internet [Bern94].

- Jpeg : Es un formato de imagen diseñado por la coorporación Joint Photographics Experts Group. Permite definir el nivel de calidad y de codificación que se quiere, por contra, sufre pérdidas a la hora de la compresión [Bust94]. Muy usado en los documentos *HTML*.

- LaTex : Es un formato de documentos basado en etiquetas. Muy usado para la escritura de textos científicos.

- Mov: Extensión de los ficheros que contienen vídeos con el formato diseñado por Apple para los Macintosh. Se reproduce mediante la aplicación QuickTime. Este programa se ha transladado a Windows, y está compitiendo con los formatos *AVI* de Microsoft [Bust94].

- Plug-in : Es un programa software que provee con servicios multimedia a las páginas web. Por ejemplo, si el plugin para el lector *PDF* de Adobe está instalado en el Netscape, y se accede a una página web que contiene un archivo *PDF*, el plug-in *PDF* se activa y permitirá ver el fichero *PDF*.

- Rmi : Remote Method Invocation. Llamada a procedimiento remoto. Es un mecanismo mediante el cual es posible llamar a funciones y métodos de objetos residentes en otras máquinas.

- Slot: 'Ranura'. Concepto similar al de 'atributo' de la orientación a objetos.

- Tabla de símbolos : Es una tabla, normalmente de tipo hash, en la que el compilador guarda la información sobre las variables del programa. Esta información la puede usar, por ejemplo, para comprobar los tipos en las expresiones, si una variable ya ha sido declarada, etc.

- TCP/IP: TCP (protocolo de control de transmisión) es un protocolo para al comunicación entre ordenadores diseñado por *ARPANET*, específicamente dirigido a tolerar el funcionamiento de una red

insegura. Asociado a *TCP*, se creó también el protocolo *IP*. *TCP* pertenece a la capa de transporte de la pila OSI, IP al de red [Tane91].

- URL: Universal Resource Location. Es la dirección Internet de un archivo.
- VRML: Virtual Reality Modelling Language. Es un lenguaje que permite describir objetos 3D y combinarlos en escenas y mundos [Hart96].
- Wav: formato de fichero de audio diseñado por Microsoft, muy usado por tanto en entornos Windows. Estos ficheros se generan mediante digitalización [Bust94].
- Web, World Wide Web: Es un sistema distribuido hipermedia basado en el modelo cliente/servidor [Quin95]. Permite a cualquier usuario de Internet acceder a documentos electrónicos distribuidos en cualquier lugar y almacenados en servidores. El intercambio de datos entre clientes y servidores se basa en una representación común de documentos, llamada *HTML*.
- Widget: Un control visual en un entorno gráfico.
- WYSIWYG: 'What You See Is What You Get' es un acrónimo que se aplica fundamentalmente a editores gráficos, en los que se va viendo mientras se edita la apariencia final del documento. Un ejemplo de un sistema *WYSIWYG* es Word, un sistema no *WYSIWYG* puede ser LaTeX.
- XML: Extensible Markup Language, es un lenguaje restringido del más general *SGML*. Sirve para definir documentos formados por datos y marcas, estas codifican una descripción de la composición los datos del documento y de su estructura lógica [XML97].

Bibliografía

[Abaq00] Abaqus home page: http://www.abaqus.com

[Aceb97] Acebes,L.F., de Prada, C. *"SIMPD : An intelligent modelling tool for dynamic processes"*. ESS97. Passau, pp. 177-181.

[Acti00] Página sobre ActiveX de Microsoft: http://www.microsoft.com/com/tech/ActiveX.asp

[Aker97] Akers, R.L. Kant, E., Randall, C.J., Steinberg, S., Young, R.L. *"Problem Solving Environments and the Solution of Partial Differential Equations"*. En Internet en: http://www-cgi.cs.purdue.edu/cgi-bin/acc/pses.cgi.

[Alfo71] Alfonseca, M. *"Desarrollo de modelos electrónicos de sistemas de regulación biológica"*, Tesis doctoral, presentada en la ETS Ingenieros de Telecomunicación. Universidad Politécnica de Madrid..

[Alfo74] Alfonseca, M. *"SIAL/71, a Continuous Simulation Compiler"*, in "Advances in Cybernetics and Systems", Ed. J. Rose, Gordon and Breach, London, Vol. 3, 1974, 1319-1340.

[Alfo97] M. Alfonseca, E. Pulido, J. de Lara, R. Orosco, *"OOCSMP: An Object-Oriented Simulation Language"*, Proc. 9th European Simulation Symposium ESS'97, pp. 44-48, 1997.

[Alfo98a] Alfonseca, M., García, F., de Lara, J., Moriyón, R. *"Generación Automática de Entornos de Simulación con Interfaces Inteligentes"*, ADIE, Octubre Diciembre 1998, pp. 5-13.

[Alfo98b] Alfonseca, M., de Lara, J., Pulido, E.,*"An object-oriented continuous simulation language and its use for training purposes"* SESP'98. 5th International Workshop on Simulation for European Space Programmes. November 98.Noordwijk, the Netherlands.

[Alfo98c] Alfonseca, M., de Lara, J., Pulido, E.,*"Educational Simulation of Complex Ecosystems in the World-Wide-Web"* . Proc. 9th European Simulation Symposium ESS'98. October, 98. Nottingham .

[Alfo98d] Alfonseca, M., de Lara, J., Pulido, E.,*"Generación semiautomática de cursos de electrónica para Internet mediante un lenguaje de simulación contínua orientado a objetos"*.TAEE'98. Madrid, Septiembre 1998 .

[Alfo98e] Alfonseca, M., Alfonseca, E., de Lara, J.,*"Compiling a simulation language in APL"*. APL Quote Quad, Vol. No. pp- . También en los proceedings de APL'98. Julio 98.Roma.

[Alfo98f] Alfonseca, M., de Lara, J., Pulido, E.,*"Semiautomatic Generation of Educational Courses in the Internet by Means of an Object-Oriented Continuous Simulation Language"*, ESM'98, Manchester.

[Alfo98g] Alfonseca, M., Carro, R., de Lara, J., Pulido, E.,*"Education in Ecology at the Internet with an Object-Oriented Simulation Language"*. EUROSIM'98, Helsinki.

[Alfo99a] Alfonseca, M., de Lara, J., Pulido, E., *"Object-oriented constructs and partial differential equations in a continuous simulation language"*, Tools Eastern Europe'99, Sofia (Bulgaria), 1-4 Jun. 1999.

[Alfo99b] Alfonseca, M., de Lara, J., Pulido, E. *"Dynamical object generation during the execution of continuous simulation models"*. Proc. ASOO'99, Third Argentine symposium on Object Orientation, Buenos Aires, 6-7 Sep. 1999. pp. 89-102.

[Alfo99c] Alfonseca, M., de Lara, J.,*"Output visualization modes in a Java generating Continuous Simulation Compiler"*, Proc. 9th European Simulation Symposium ESS'99, Erlangen, 26-28 Oct. 1999, pp. 159-163.

[Alfo99d] Alfonseca, M., de Lara, J., Pulido, E. *"Semiautomatic Generation of Web Courses by Means of an Object-Oriented Simulation Language"*, special issue of "SIMULATION", Web-Based Simulation, Vol 73, num. 1, Julio 1999, pp. 5-12.

[Alfo99e] Alfonseca, M., de Lara, J. *"Visualización en Internet de cursos basados en Simulación Continua"*, Interacción hombre máquina, curso de verano de la Universidad de Castilla La Mancha. Puertollano, Octubre de 1999.

[Alfo99f] Alfonseca, M., de Lara, J. *"Integración de Simulación y Multimedia en cursos generados automáticamente para Internet "*, proc. CONIED'99. Puertollano, Noviembre de 1999.

[Alfo00a] Alfonseca, M., de Lara, J. *"Distributed simulation of systems based on partial differential equations at the Internet"*, proc. 16th IMACS World Congress, Laussanne.

[Alfo00b] Alfonseca, M., de Lara, J. *"Distributed simulation of Ecosystems for the Internet"*, proc. 16th IMACS World Congress, Laussanne.

[Alfo00c] Alfonseca, M., de Lara, J. *"Automatic generation of a course on electronics with associated documentation"*, proc. EUROMEDIA 2000, Antwerp, Belgium.

[Ambo99] Amborski, K., Pester, M., Scholz, H., *"Digital simulation of Continuous Systems"*, Proc. EUROMEDIA '99. pp. 207-209.

[ASCE91] ASCEND IV home page : http://www.cs.cmu.edu/~ascend/Home.html.

[Avia97] Aviation Industry CBT Committee Computer Managed Instruction. 1997. *Computer Managed Instruction Guidelines and Recommendations*, AGR 006, Version 1.1, AICC. http://www.aicc.org/agr006.htm.

[Baeh87] Baehmann,P.L. et al., *"Robust, geometrically based, automatic two-dimensional mesh generation"*, International Journal for Numerical Methods in Engineering, 24, 1043-1048 (1987).

[Berg97] Berg, D.J.; Fritzinger, J.S., *1997."Advanced Techniques for Java Developers"*, John Wiley&Sons,Inc.

[Bern94] Berners-Lee, T., Connolly, D. "Hypertext Markup Language Specifications – 2.0", Internet

[BGC00] Bond Garph Compendium, en Internet en : http://www.eng.gla.ac.uk/bg/

[Blit90] *"Arrays in Blitz++"*, SCOPE'98, vol. 1505 of Lecture Notes in Computer Science, 1998.

[Bowy81] Bowyer, A., *"Computing Dirichlet tessellations"*, The Computer Journal, 24, 162-166.

[Brau93] Braun, M. 1993. *"Differential equations and their applications"*. Springer-Verlag, 4th. Edition.

[Bred98] Bredeweg, B., Winkels, R. 1998. *"Qualitative Models in Interactive Learning Environments: an Introduction"*. Special Issue of Interactive Learning Environments: the Use of Qualitative Reasoning Techniques in Interactive Learning Environments. pp. 1-18, vol. 5. 1998.

[Bree85] Breedveld, P.C., *"Multibond graph elements in physical systems theory"* Journal of the Franklin Institute, 319 (1/2):1-36, January/February 1985.

[Bris99] Bris Data home page : http://www.brisdata.se

[Broe96] Broenink, J.F., Weustink, P.B.T., *"A Combined-System Simulator for mechatronic Systems"*, Proc ESM'96, Budapest. pp.225-229.

[Broe97] Broenink, J.F., *"Bond-Graph Modeling in Modelica"*, Proc. ESS97, Passau. pp. 137-141.

[Broe99] Broenink, J.F., *"Object-Oriented modeling with bond graphs and Modelica"*, ICBGM'99, International Conference on Bond Graph Modeling and Simulation, parte de la WMC'99, Western MultiConference, San Francisco, CA. Simulation Series, Vol 31, n° 1, pp. 163-168.

[Broo97] Brooks, D. 1997. *"Web-teaching"*. Plenum Press.

[Buss96] Buss, A.H., Uhrmacher, K.A. 1996. *"Discrete Event Simulation on the World Wide Web using Java"*. Proc. 1996 Winter Simulation Conference, pp. 780-785.

[Bust94] Bustos, I. 1994. *"Multimedia"*. Biblioteca Informática de PC Magazine.

[Carr99] Carro, R.M., Pulido, E., Rodríguez, P. 1999.*"Dynamic generation of adaptive Internet-based courses"*. Journal of Network and Computer Applications (1999) 22, pp. 249-257.

[Cell86] Cellier, F.E. (Ed.), 1986, *"Languages for Continuous System Simulation"*, Proc. SCS Conference on Continuous System Simulation Languages, San Diego, California.

[Cell91] Cellier, F.E. 1991, *"Continuous System Modeling"*, Springer Verlag.

[Cour28] Courant, R., Friedrichs, K.O., Lewy, H. 1928. *"Über die partiellen differenzengleichungen der mathematischen physik"*. Mathematische Annalen, 100:32-74.

[Cube98] Cubert, R.M., Fishwick, P.A. 1998. *"OOPM: An Object-Oriented Multimodeling and Simulation Application Framework"*. SIMULATION, Vol. 70. no. 6. pp 379-395.

[Cran47] Crank, J., and Nicolson, P., 1947. *"A practical method for numerical integration of solutions of partial differential equations of heat-conduction type"*, proc. of the Cambridge Philosophical Society, 43:50-67.

[Chan91] Chandrashekar,M., Savage, G.J., *"Engineering systems – analysis, design and control"*, Systems Design Engineering, University of Waterloo (Ontario), 1991.

[Chew89] Chew, P. L. 1989. *"Guaranteed-Quality Triangular Meshes"*, TR 89-983, Department of Computer Science, Cornell University, Ithaca, NY, April 1989.

[Chri83] Christy, D. and H. Watson. 1983. *"The Application of Simulation : A Survey of Industry Practice"*, Interface, Vol.13, n°5, pp.47-52.

[Dahl66] Dahl, O.J. and K. Nygaard, 1966 *"SIMULA – An ALGOL-Based simulation language"*, Communications of the ACM, 9:9, 671-678.

[deCo97] de Cogan,D., de Cogan A., 1997, *"Applied Numerical Modelling for Engineers"*, Oxford Science Publications. Textbooks in electrical and electronic engineering.

[deJo91] de Jong, T. (editor). 1991. *"Computer simulations in an instructional context"*. Education and Computing (especial issue), no. 6.

[deLa99] de Lara, J., Alfonseca, M. *"Simulating Partial Differential equations in the World-Wide Web"* Proceedings EUROMEDIA'99. pp-45-52, Munich, Abril 1999.

[Dela34] Delaunay,N.; 1934 *"Sur la Sphere"* Vide. Izvestia Akademia Nauk SSR, VII Seria, Otdelenie Matematicheskii i Estestvennyka Nauk Vol 7 pp.793-800.

[Dham97] Dhamman, J.S., Fujimoto, R.M., Weatherly, R.M. 1997. *"The Department of Defense High Level Architecture"*. Proc. 1997 Winter Simulation Conference.

[Díaz99] Diáz-Calderón, A., Paredis, C.J.J, Khosla, P.K., *"A composable simulation environment for mechatronic systems"*, Proceedings of ESS'99, pp. 142-148, Erlangen, Octubre 1999.

[Digi90] Digitalk Inc., 1990 *"Smalltalk/V PM"*, Digitalk Inc., Los Angeles.

[Dijk65] Dijkstra, E. W. 1965. *"Cooperating Sequential Processes"*. Technical Report EWD-123, Technologcal University, Eindhoven, Holanda.

[DMSO99] Defense modelling and simulation office home page: http://www.dmso.mil/

[Drea00] DreamWeaver home page : http://www.macromedia.com/software/dreamweaver/

[Drum00] Drumbeat home page : http://www.elementalsoftware.com

[Dymo99] Dymola home page : www.dynasim.se

[Ecos99] Ecosim home page : http://www.empre.es

[Elmq78] Elmqvist, H., 1978, *"A Structured Model Language for Large Continuous Systems"*. CODEN: LUTED2/TFRT-1015, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

[Elmq97] H. Elmqvist, S.E. Mattson, *"An introduction to the Physical Modeling Language Modelica"*, Proc 9th European Simulation Simpussium ESS97, pp. 110-114.

[Engl96] Engler, D.R., Hsieh, W.C., Kaashoek, *"'C: A language for high-level, efficient, and machine-independent dynamic code generation"*, in POPL'96. pp. 131-144.

[Erik98] Eriksson, H.E., Penker, M. 1998. *"UML Toolkit"*. John Wiley&Sons.

[Fahr70] Fahrland, D. 1970. *"Combined Discrete Event and Continuous Systems Simulation"*, Simulation, Vol. 13, nº 2, pp. 61-72.

[Fede99] Página de Federico García: http://www.ii.uam.es/~fede/research.html

[Fiel88] Field, D.A. *"Laplacian smoothing and Delaunay triangulations"*. Comm. Applied Numer. Meth., 4:709-712, 1988.

[Fili96] Filipak, M. 1996. *"Mesh Generation v1.0"*, Technical paper at Edinburgh Parallel Computing Centre.

[Fish88] Fishwick, P.A. 1988. *"The Role of Process Abstraction in Simulation"*, IEEE Transactions on Systems, Man & Cybernetics, Vol.18, nº1, pp. 18-39.

[Fish92] Fishwick, P.A., Zeigler, B.P. 1992. *"A Multimodel Methodology for Qualitative Model Engineering"*. ACM Transactions on Modeling and Computer Simulation, Vol.2, No.1, pp. 52-81.

[Fish93] Fishwick, P.A., Lee, K. 1993. *"Two Methods for Exploiting Abstraction in Systems"*. AI, Simulation and Planning in High Autonomous Systems, pp. 257-264.

[Fish96] Fishwick, P.A. 1996. *"Web-based Simulation : Some personal observations."*, proc. 1996 Winter Simulation Conference, Coronado, CA.1996.

[Fish98] Fishwick, P.A. 1998. *"Issues with Web-Publishable Digital Objects"*, in proc. of SPIE: Enabling Technologies for Simulation Science II, pp. 136-142.

[Floy97] Floyd, T.H. 1997. *"Digital fundamentals"*, 6th edition, Simon & Schuster International Group

[Forr61] Forrester, J.S. 1961. *"Industrial Dynamics"*. MIT Press, Cambridge, Mass.

[Fox98] Fox, G.C., Furmanski, W., Nair, S., Ozdemir, H.T., Ozdemir, Z.O., Pulikal, T.A., *"WebHLA – An interactive Multiplayer Environment for High Performance Distributed Modeling and Simulation"*, in proc. of the 1999 International Conference on Web-Based Modeling and Simulation, pp. 163-168.

[Fron00] Microsoft FrontPage home page : http://www.microsoft.com/frontpage

[Garc98] García, F., Contreras, J. Rodríguez, P. and Moriyon, R., 1998. *"Help generation for task based applications with HATS"*, IFIP Working Conference on Engineering for Human-Computer Interaction, Creta.

[Garc99] García,P, Gómez Skarmeta,A., Meseguer,J.J, Ibañez,J. *"VR-CLASS : A Multi-User VRML teaching environment"*, proc. EUROMEDIA'99, pp.3-10.

[Gayl95] Gaylord, R.J., Wellin, P.R., *"Computer Simulations with Mathematica"*, Telos, The Electronic Library of Science, Santa Clara, CA.

[Geor91] George, P.L., *"Automatic mesh generation: application to finite element methods"*, Wiley, 1991.

[GID99] Gid home page en : http://gid.cimne.upc.es

[GMCL00] Generative Matrix Computation Library home page: http://nero.prakinf.tu-ilmenau.de/czarn/gmcl

[GNA97] GNA The Globewide Network Academy. 1997. http://www.gnacademy.org.

[Gour70] Gourlay, A.R. 1970, J. Inst. Maths. Appls., vol. 6, pp. 375-384.

[Hans73] Hansen, P. B. 1973. *"Operating System Principles"*. Prentice-Hall, Englewood Cliffs, New Jersey.

[Hart96] Hartman, J., Wernecke, J. 1996. *"The VRML v2.0 handbook"*. Addison-Wesley.

[Heal97] Healy, K.J., Kilgore, R.A. *"Silk: A Java-Based Process Simulation Language"* proc. 1997 winter simulation conference, Atlanta, pp. 475-482.

[Hild68] Hildebrand, F.B., 1968, *"Finite-Difference equations and Simulation"*, Prentice-Hall, N.J.

[HLA99] HLA Home page : http://hla.dmso.mil/

[Hoar74] Hoare, C.A.R. 1974. *"Monitors: An Operating System Structuring Concept"*. Communications of the ACM, Vol. 17, No. 10. pp. 549-557; Erratum in Communications of the ACM, Vol 18, no. 2. pag. 95.

[Home00] HomeSite home page: http://www.allaire.com

[HotD00] HotDog home page : http://www.sausage.com

[HotM00] HoTMetaL home page : http://www.softquad.com

[Howe98] Howell,F., McNab,R. 1998. *A Discrete Event Simulation Library for Java.* In Fishwick P., Hill D., Smith R., Ed: Proceedings of the 1st International Conference on Web-based Modeling and Simulation, SCS San Diego.

[Hsua97] Hsuan-Cheng, Lin 1997, *"JAVAMESH- A two dimensional triangular mesh generator for finite elements"*, submited for MS at Pittsburgh University.

[Hugh87] Hughes, Thomas J.R; 1987, *"The Finite Element Method : Linear Static and Dynamic Finite Element Analysis"*, Prentice Hall International.

[IBM72] IBM Corp. 1972. Continuous System Modelling Program III (CSMP III) and Graphic Feature (CSMP III Graphic Feature) General Information Manual. IBM Canada, Ontario, GH19-7000.

[JáJá92] JáJá, J. 1992. *"An introduction to parallel algorithms"*, Addison-Wesley.

[Java99] http://java.sun.com

[JGra99] Interim Java Grande Forum Report. *"Java Grande Forum"*, Technical Report JGF-TR-4, see : http://www.javagrande.org/report.htm .

[JMF99] Java Media Framework page : http://java.sun.com/products/java-media/jmf/2.0

[Jone96] Jones,J.,Roberts,S., *"Design of Object Oriented Simulations in C++"*, proc. 1996 Winter Simulation Conference.

[Karm98] Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Reynders, J., Smith, S., Williams, T., *"Array design and expression evaluation in POOMA II"*, ISCOPE'98, vol. 1505, Sipringer-Verlag, 1998. Lecture Notes in Computer Science.

[Karn90] Karnopp, D. *"Bond Graph Models for Electrochemical Energy Storage: Electrical, Chemical and Thermal Effects"*, Journal of the Franklin Institute, Vol 324, 1990, pp. 983-992.

[Kasc79] Kascic, M.J. 1979. *"Vector Processing on The Cyber 200"* Infotech State of the Art Report "Supercomputers", Infotech International Ltd, Maidenhead, U.K., pp. 1-38.

[Kirk99] Kirkup, S., *"An Introduction to the Boundary Element Method"*, at http://www.soundsoft.demon.co.uk/laplace.htm

[Klei98] Klein, U.S., Strassburger, Beikrich, J. 1998. *"Distributed Simulation with JavaGPSS based on the High Level Architecture"*. Proc of the 1998 SCS International Conference on Web-Based Modeling and Simulation, pp. 85-90.

[Knup93] Knupp, P. and Steinberg, S., *"Fundamentals of grid generation"*, Boca Raton, CRC Press, 1993 : ver http://math.unm.edu/stanly/me/Fortran.shar.

[Lapi82] Lapidus, L; Pinder, G.F. 1982. *"Numerical Solution of Partial Differential Equations in science and engineering"*. John Wiley&Sons.

[Laws77] Lawson, C.L.,1977 *"Software for C^1 Surface Interpolation"*, Mathematical Software III, pp.161-194.

[Lee97] Lee, R.C., Tepfenhart, W.M. 1997. *"UML and C++, a practical guide to object-oriented development"*, Prentice-Hall.

[Lega80] Legasto A.A. Jr., Forrester, J.W., Lyneis, J.M. editors, *"Systems Dynamics"*, North Holland, 1980.

[Leon94] Leone, M., Lee. P., *"Lightweight run-time code generation"*, Proceedings 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Base Program Manipulation, Technical Report 94/9, Department of Computer Science, University of Melbourne, Junio 1994, pp. 97-106.

[Long99] Long, J., Spencer, P., Springmeyer, R. 1999. *"SimTracker - Using the Web to Track Computer Simulation Results"*. 1999 International Conference on Web-Based Modeling and Simulation.

[Lore94] Lorenz F., 1994, *"Comments on the Neutral Model Format"*, presented to the TC4.7 NMF Ad Hoc Subcommittee at the ASHRAE winter meeting 1994.

[Lore99] Lorenz Simulation home Page : http://www.lorsim.be

[Macr00] Macromedia Director home page : http://www.macromedia.com

[Maly98] Maly, K., Overstreet, C.M., González, A., Denbar, M., Cutaran, R., N. Karunaratne, C. J. Srinivas. 1998. *"Use of Web Technology for Interactive Remote Instruction"*. Proceedings of the Web'97 Conference. En Internet : http://www7.scu.edu.au/programme/posters/1855/com1855.htm

[Matl99] Matlab home page : http://www.mathworks.com

[Mend00] Mendelson, E. *"Web authoring tools"*, PC Magazine, OnLine, en Internet en: http://www.zdnet.com/pcmag/features/htmlauthor/_intro.htm

[Mins85] Minsky, M. 1985. *"Model, Minds, Machines"* Proc. IFIPS, AFIPS Press, Montvale, NJ, pp. 39-45.

[Mode99a] Modelica home page : http://www.mathworks.com.org

[Mode99b] ModelManager/Model Maker home page : http://www.modelmanager.com/

[Mons97] Monsef, Y. *"Modelling and Simulation of Complex Systems"*, SCS Int., Erlangen, 1997.

[Mora00] Mora, M.A., Moriyón, R. 2000. "Collaborative History Management for Chess and Tutoring Systems". First Technical Workshop of the Computer Engineering Departament. UAM. En Internet en : http://www.ii.uam.es/eng/research/workshop/mora.ps

[Most97] Mosterman, P.J., *"Hibrid dynamic systems : A hybrid bond graph modeling paradigm and its application in diagnosis"*. Ph.D thesis, Vanderbilt University, Nashville, Tenessee.

[MS197] MS1, Modelling System 1 version 3.2 reference manual, Diciembre 1997. Lorenz Simulation, S.A.

[Myer97] Brad Myers et al. *"The Amulet v3.0 reference Manual"* Carnegie Mellon University School of Computer Science Technical Report no. CMU-CS-95-166-R2 and Human Computer Interaction Institute Technical Report CMU-HCII-95-102-R2, Marzo 1997.

[Nair96] Nair, R.S., Miller, J.A., Zhang, Z. 1996. *"Java-Based Query Driven Simulation Environment"*. proceedings of the 1996 Winter Simulation Conference, pp. 786-793.

[Nata94] Nataff, J.M. 1994 *"Translator from Neutral Model Format to SPARK"*, draft paper presented to the TC 4.7 NMF Ad Hoc Subcommittee at the ASHRAE winter meeting 1994.

[NE/NA00] NE/NASTRAN Noran Engineering Home page: http://www.noraneng.com

[Neto00] NetObjects home page : http://www.netobjects.com

[NMGS99] NMGS home page en : http://members.xoom.com/nmgs_list/

[Open97] OpenAMULET home page : http://www.scrap.de/html/amulet.htm

[Otto92] Ottosen, N., Peterson, H. 1992. *"Introduction to the finite element method"*. Prentice Hall.

[Page97] Page,E.H., Moose Jr., R.L., Griffin,S.P. *"Web-Based simulation in Simjava using Remote Method Invocation"*. proc. 1997 winter simulation conference, pp. 468-474, 1997.

[Page98a] Page, E.H. 1998. *"The Rise of Web-Based Simulation: Implications for the High Level Architecture"*. Proc. of the 1998 SCS International Conference on Web-Based Modeling and Simulation, pp. 123-128.

[Page98b] Page,E.H., Griffin,S.P. 1998. *"Transparent Distributed Web-Based Simulation using Simjava"*. In Fishwick P., Hill D., Smith R., Ed: Proceedings of the 1st International Conference on Web-based Modeling and Simulation, SCS, San Diego.

[Page99a] Page, E.H., Opper, J.M. 1999. *"Investigating the Application of Web-Based Simulation Principles within the Architecture for a Next-Generation Computer Generated Forces Model"*. to appear in: Future Generation Computer Systems, Elsevier Science Publishing.

[Page99b] Page, E.H., Opper, J.M. 1999. *"Observations on the Complexity of Composable Simulation"*. Proceedings of the 1999 Winter Simulation Conference, pp. 553-560.

[Page99c] Page, E.H., 1999. *"Beyond Speedup: PADS, the HLA and Web-Based Simulation"* .Proceedings of the 13th Workshop on Parallel and Distributed Simulation, R.M. Fujimoto and S. Turner, Eds., pp. 2-9.

[Page00] Page E.H. Buss, A., Fishwick, P.A., Healy, K., Nance, R.E., *"Web-Based Simulation: Revolution or Evolution?"*, to appear in ACM Transactions on Modeling and Computer Simulation.

[Payn61] Paynter, H.M. *"Analysis and design of engineering systems"*, The MIT press, Cambrifge, Massachusetts, 1961.

[Peac55] Peacemann, D.W., Rachford, H.H.. 1955. *"The numerical Solution of Parabolic and Elliptic Differential Equations"*, Journal of the Society for Industrial and Applied Mathematics, 3:28-41.

[Pear93] Pearson, J.E. 1993. *"Complex Patterns in a Simple System"*. Science, Vol 261, 9 July 1993 : 189-192.

[Pera87] Peraire, J., Vahdati, M., Morgan, K., Zienkiewicz, O.C., *"Adaptative remeshing for compressible flow calculations"*, Journal of Compurational Physics, 72, 449-466 (1987).

[Pete91] Pereson, J.L., Silberschatz, A. *"Sistemas Operativos, Conceptos Fundamentales"*. Editorial Reverté

[Pidd93] Pidd, M. *"Computer Simulation in Management Science"* 3rd Edition, John Wiley & Sons.

[Poew99] PowerSim home page: http://www.powersim.no/

[Pool77] Poole, T.G., Szymankiewicz, J.Z. 1977. *"Using Simulation to Solve Problems"*. McGraw-Hill, U.K.

[Quin95] Quint, V., Roisin, C., Vatton, I. "A Structured Authoring Environment for the World-Widw Web", Proceedings of WWW'95. En Internet : http://www.igd.fhg.de/archive/1995_www95/proceedings/papers/84/EditHTML.html

[Redd93] Reddy, J.N, *"An introduction to the Finite Element Method"*, 2nd edition.McGraw-Hill, 1993.

[Rich94] Richmond, B., 1994. *"System dynamics/Systems Thinking: Let's Just Get On With it"*, 1994 International Dynamics Conference in Sterling, Scotland.

[Roach72] Roache, P.J. 1972 *"Computational Fluid Dynamics"*. Hermosa Publisers, Albuquerque, NM.

[Rodr70] Rodríguez de la Fuente, F. et al. 1970, *"Enciclopedia Salvat de la Fauna"*, Salvat.

[Rodr97] Rodríguez, P., García, F., Contreras, J., Moriyón, R *"Parsing Techniques for User-Task Recognition"*.5th International Workshop on Advances in Functional Modeling of Complex Technical Systems, Paris (France),Julio 1997.

[Rose83] Rosenberg, R.C., Karnopp D.C, *"Introduction to physical system dynamics"*, Mac Graw-Hill Book Company, New York 1983.

[Rosk72] Rosko, J. *"Digital Simulation of Physical Systems"*, Addison-Wesley. 1972.

[Rour94] O'Rourke, *"Computational Geometry in C"*, Cambridge University Press, 1994.

[Rubi98] Rubinstein, I. ; Rubinstein, L. 1998. *"Partial differential equations in classical mathematical physics"*. Cambridge University Press.

[Rupp92] Ruppert, J., 1992 *"A New and Simple Algorithm for Quality 2-Dimensional Mesh Generation"*. Technical Report UCB/CSD 92/694, University of California at Berkeley, Berkeley California.

[Sahl89] Sahlin,P., Sowell, E., 1989 *"A Neutral Format for Building Simulation Models"*, conference proceedings Building Simulation'89, IBPSA, Vancouver, Junio, 1989.

[Sahl96] Sahlin, P., Bring,A., Sowell, E.F., 1996, *"The Neutral Model Format for Building Simulations"*, Version 3.02, approved by the ASHRAE NMF Committee at the Atlanta meeting, Febrero 1996. En internet en : ftp://mailbase.ac.uk/pub/lists-f-j/ibpsa-nmf

[Sahl99] Sahlin, P. 1999, *"A tool for data mapping from design products to object-oriented simulation models"*, 11th European simulation symposium ESS'99, Simulation in industry, pp. 18-22.

[Scha95] Schank, R.C, Cleary, C. 1995. *"Engines for Education"*. Lawrence Erlbaum, Hillsdale, New Jersey.

[Schi90] Schildt, H. 1990, *"Programación en Turbo C"*, Borland Osborne/Mc Graw Hill.

[Schm99a] Schmid, Ch. *"A Remote Library Using Virtual Reality on the Web"*, SIMULATION, special issue : Web-Based Simulation, Vol 73 Julio 1999. pp: 13-21.

[Schm99b] Schmid, Ch. *"Simulation and virtual reality for education on the web"*, proc EUROMEDIA'99, pp:181-188.

[Schw95] Schwetman,H. *"Object-Oriented simulation modeling with C++/CSIM17"* proc. 1995 winter simulation conference.

[Schu97] Schutte. 1997. *Virtual Teaching in Higher Education: The New Intellectual Superhighway or Just Another Traffic Jam?*. http://www.csum.edu/sociology/virexp.htm

[SEI00] Página sobre COM del Carnegie Mellon University Software Engineering Institute: http://www.sei.cmu.edu/str/descriptions/com_body.html

[Serb98] Serbedzija, N.B. 1997. *"The Web supercomputing environment"*. Proceedings of the WWW'97. En Internet en : http://www7.scu.edu.au/programme/posters/1838/com1838.htm

[Shih83] Shih, T.M. (ed), 1981. *"A Survey of Finite Difference Schemes for Parabolic Pratial Differential Equations"*, Ames,W.F, Proc. of the 2nd National Symposium on Numerical Methods in Heat Transfer. Publicado en en *"Numerical Properties and Methodologies in Heat Transfer"*, Springer-Verlag.

[Sieg96] Siegel, J. 1996. "CORBA Fundamentals and Programming". John Wiley & Sons.

[Siek98] Siek, J.G., Lumsdaine, A. 1998. *"A rational approach to portable high perfomance:The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library"*, in Parallel Object Oriented Scientific Computing, ECOOP.

[Sier99] Sierra, A., Alfonseca, M. 1999. *"Programación en C/C++"*. Anaya Multimedia.

[Spec76] Speckhart, F. 1976. *"A Guide to using CSMP"*. Prentice Hall Inc., Englewood Cliffs, N.J.

[Spri82] Spriet, J.A., Vansteenkste, G.C. 1982. *"Computer-aided Modelling and Simulation"*. Academic Press, International Lecture Series in Computer Science.

[Srid97] Sridharan. P. 1997. *"JavaBeans Developer's resource"*. Prentice Hall.

[Stau96] Stauffer, D., Stanley, H.E. 1996. *"From Newton to Mandelbrot"*2nd Edition. Springer.

[Stel99] STELLA home page : http://www.hps-inc.com

[Stic97] Stichnoth, J., Gross, T. 1997. *"Code composition as an implementation language for compilers"*, in USENIX Conference on Domain-Specific Languages.

[Stou93] Stour, J. ; Bulirsch, R. 1993. *"Introduction to Numerical Analysis"*. Springer-Verlag, 2nd Edition.

[Stra92] Strauss, W.A. 1992. *"Partial Differential Equations, an Introduction"* John Wiley&Sons.

[Stri89] Strikwerda, J.C. 1989. *"Finite difference schemes and partial differential equations"*. Chapman & Hall; New York.

[Stro97] Stroupstrup, B., 1991-1997. *"The C++ Programming Language"*, Addison-Wesley, Reading (Mass).

[Stru00] Structural Design & Analysis Corp. (COSMOS/M) home page: http://www.cosmosm.com/

[Tane91] Tanenbaum,A.S., 1991, *"Redes de ordenadores"*, 2ª edición. Prentice Hall.

[Tane93] Tanenbaum,A.S., 1993, *"Sistemas operativos modernos"*, Prentice Hall.

[Tayl99] Taylor, J., AME home page: http://helios.bto.ed.ac.uk/ierm/ame/

[Thom85] Thompson, J.F.; Warsi, Z.U.A. ; Mastin, C.W. 1985. *"Numerical Grid Generation"*, Elsevier Science Publishing CO.Inc. In internet at:http://WWW.ERC.MsState.Edu/education/gridbook.

[Thom89] Thoma, J. U., 1989. *"Simulation by bond-graphs – Introduction to a graphical method"*, Springer-Verlag.

[Thom97] Thomson Publishing. 1997. *Internet Distance Education with Visual C++.* http://www.thomson.com/microsoft/visual-c/teacher.html

[Thom99] Thomson. J.F. *"A reflection on grid generation in the 90s: Trends, Needs, and Influences"*.

[Törn81] Törn, A.A., 1981. *"Simulation Graphs: a General Tool for Modelling Simulation Design"*. Simulation, Vol. 37, nº 6, pp.187-194.

[Veld98] Veldhuizen, T.L., Gannon, D. *"Active Libraries: Rethinking the roles of compilers and libraries"*

[Volt31] Volterra, V., 1931, *"Leçons sur la Théorie Mathèmatique de la Lutte pour la Vie"*, Gauthier-Villars, Peris.

[Wats81] Watson, David F., 1981 *"Computing the Delaunay Tesselation with Application to Voronoi Polytopes"*, The Computer Journal, Vol 24(2) pp. 167-172.

[Webe95] Weber, K., *"Title: Chapter 6, In which Pooh proposes improvements to Web authoring tools, having seen said tools for the Unix platform"*. Proceedings of the WWW'95. En internet en : http://www.igd.fhg.de/archive/1995_www95/proceedings/papers/104/katew.paper.html

[Well79] Ellstead, P.E., *"Introduction to physical system modelling"*, Academic Press, London, 1979.

[Wolf94] Wolfram, S. 1994. *"Cellular automata and Complexity"*. Addison Wesley

[XML97] Especificación de *XML* v1.0. http://www.w3.org/TR/WD-xml-lang.html

[Yoha69] Yoahan Chu, 1969. *"Digital simulation of continuous systems"*, MacGraw Hill.

[Zeig76] Zeigler, B. 1976. *"Theory of Modelling and Simulation"*. John Willey & Sons, N.Y.

[Zeig90] Zeigler, B, Ahang, G. 1990. *"Mapping Hierarchical Discrete Event Models to Multiprocessor Systems: Concepts, Algorithm, and Simulation"*. Journal of Parallel and Distributed Computing, Vol.9, pp.271-281.

[Zien71] Zienkiewicz, O.C., Phillips, D.V. *"An automatic mesh generator scheme for plane and curved surfaces by isoparametric coordinates"*, International Journal for Numerical Methods in Engineering, 3, 519-528, (1971).

[Zien89] Zienkiewicz, O.C ; Taylor, R.L. 1989. *"The Finite Element Method"*, 4th edn, vol. I, McGraw-Hill; New York.

[Zink96] Zinkovsky, A.V., Sholuha, V.A., Ivanov, A.A. 1996. *"Mathematical Modelling and Computer Simulation of Biomechanical Systems"*. World Scientific.

Reunido el tribunal que suscribe en el día de la fecha, acordó calificar la presente Tesis doctoral con SOBRESALIENTE CUM LAUDE

Madrid, 2-6-00