Universidad de Las Palmas de Gran Canaria

Instituto Universitario de Sistemas Inteligentes y
Aplicaciones Numéricas en Ingeniería

Tesis Doctoral

Implementación de un algoritmo de refinamiento/desrefinamiento para mallas de tetraedros

José María González Yuste

Las Palmas de Gran Canaria, mayo de 2007

Universidad de Las Palmas de Gran Canaria

Programa de Doctorado en Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Instituto Universitario de Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería

Tesis Doctoral

Implementación de un algoritmo de refinamiento/desrefinamiento para mallas de tetraedros

Autor | Director | Director

José Mª González Yuste | Rafael Montenegro Armas | José Mª Escobar Sánchez

Las Palmas de Gran Canaria, mayo de 2007

A Mila y Javier

Agradecimientos

Hay muchas personas que me han apoyado y han contribuido para que este trabajo llegara a buen término.

En primer lugar, quiero mencionar a mi director Rafael Montenegro. Han sido innumerables las tardes que hemos pasado comentando infinidad de temas (y no sólo de investigación). Su paciencia conmigo y sus actos de fe cuando yo le proponía alguna locura no tienen precio.

También quiero agradecer el entusianmo de José Mª Escobar, mi otro director, y de Gustavo Montero, director de la división GANA del IUSIANI. José Mª siempre anda buscando algo nuevo, siempre proponiendo alguna aplicación... debería descansar un poco. Gustavo siempre aportaba alguna solución cuando nos atascábamos en un problema y nos daba una salida fácil y elegante.

A mi compañero (ya doctor) Eduardo Rodríguez Barrera, le debo más de lo que puedo agradecerle en estas líneas. Me ha facilitado muchísimo las cosas y siempre ha estado a mi disposición cuando lo he necesitado.

A mi mujer, Mila... por fin! Siempre ha estado animándome, empujándome y hasta persiguiéndome para que continuara con la labor investigadora. Si mi ocupación laboral y familiar me llevaban el 110 % del tiempo, has conseguido aligerarme muchísimo la carga para que siguiera adelante. Gracias por tu paciencia y tu comprensión en todas esa jornadas atípicas en las que el dormir era un sueño. Y por encargarte de nuestro hijo, Javier, que antes de cumplir los dos años ya distinguía perfectamente lo que era un triángulo. Afortunadamente aún no sabe lo que es un tetraedro.

También quiero mencionar a mis padres, hermanos, familiares y amigos que han estado siempre interesándose y animándome para que concluyera este trabajo. Gracias a todos.

Esta tesis ha sido desarrollada en el marco de los proyecto subvencionados por el Ministerio de Ciencia y Tecnología y FEDER, REN2001-0925-C03-02/CLI titulado *Modelización numérica de transporte de contaminantes en la atmósfera*, y CGL2004-06171-C03-02/CLI titulado *Modelización y simulación numérica de campos de viento orientados a procesos atmosféricos*.

Índice general

Índice de figuras

Índice de Tablas

Índice de Algoritmos

Capítulo 1

Introducción

Durante el siglo XX pocos métodos de aproximación como el de los *elementos finitos* han tenido tanto impacto en la teoría y práctica de los métodos numéricos [Tinsley Oden, 1990]. Los trabajos de [Hrennikoff, 1941] y [Courant, 1943] sentaron las bases del método que actualmente se utiliza. Aunque de forma diferente, ambos autores realizan una partición de un dominio continuo en un conjunto de subdominios discretos.

El desarrollo del método como tal se produjo en la segunda parte de los años 50 para el análisis de estructuras y en la aeronáutica. [Wilson y Clough, 1999] recoge las aplicaciones que en los años 60 se le dieron en ingeniería civil desde la Universidad de Californa (Berkeley). El método fue demostrado con un riguroso análisis matemático con la publicación de [Strang y Fix, 1973], y desde entonces ha sido empleado en multitud de problemas numéricos para la modelización de sistemas físicos en un amplio rango de disciplinas, como electromagnetismo o dinámica de fluidos.

La partición que se realiza del dominio de estudio en elementos finitos se denomina discretización. En cada elemento se distinguen una serie de puntos representativos llamados *nodos*. Una *malla* será el conjunto de todos los nodos considerando sus relaciones de adyacencia. Esta malla deberá ser elaborada por un *generador de mallas* en un proceso previo a la aplicación del método.

Una vez obtenida una solución inicial mediante el método de elementos finitos, lo que se busca es aplicar un *procedimiento adaptativo* que ajuste automáticamente la malla en las zonas en las que se necesite mejorar la solución numérica y con un esfuerzo mínimo. Las técnicas más usuales son:

- Refinamiento en h, mediante la adición o supresión de elementos sobre la

malla original

- Refinamiento en r, reubicando o moviendo los elementos de la malla
- Refinamiento en p, modificando el grado del polinomio en cada elemento finito

Estas estrategias podría usarse solas o combinadas. El refinamiento en r no es capaz, por sí mismo, de ajustar una solución numérica con una precisión deseada. Si la malla es muy gruesa no se podrá alcanzar un grado de precisión suficiente sin añadir elementos. Por otro lado, el refinamiento en p puede obtener buenos resultados en ciertos tipo de problemas (con mallas bien adaptadas) ya que no tiene un coste computacional alto ni necesita añadir elementos en la malla, además de presentar una rápida convergencia.

El refinamiento en h es el más extendido y más apliamente aceptado, aunque en ocasiones se emplea combinado con el refinamiento en p (hp-refinement). La idea del refinamiento en h es realizar particiones cada vez más finas de la malla dividiendo elementos sucesivamente. Cuando el problema presenta singularidades, un refinamiento en h localizado en dichas zonas hará que el método de elementos finitos converja en una solución numérica más precisa. Por otra parte, mediante el proceso inverso (desrefinamiento), se puede simplificar el sistema suprimiendo elementos que no aporten información o que ésta pueda obtenerse muy fácilmente a partir de los datos calculados.

1.1. Estado del arte

En la actualidad, la mayor parte de los programas que utilizan el método de elementos finitos se apoya en técnicas adaptables basadas en una estimación del error cometido con la solución numérica, o al menos en indicadores de error fiables que señalen los elementos que deben ser refinados o desrefinados en la malla.

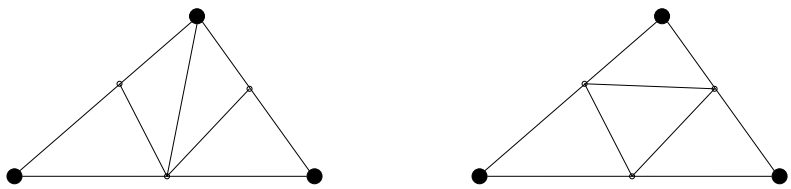
En generación de mallas adaptables se pueden considerar dos aspectos diferentes: la discretización del dominio atendiendo a su geometría o a la solución numérica. Existen muchas formas de abordar estos aspectos. La primera cuestión es: ¿mallas estructuradas o no estructuradas?. En este sentido, está claro que el uso de mallas no estructuradas proporciona más flexibilidad a la hora de mallar geometrías complejas utilizando un número óptimo de nodos. En este

caso, los métodos más clásicos para la obtención de triangulaciones tridimensionales se basan fundamentalmente en algoritmos de avance frontal [Löhner y Parikh, 1998], o en algoritmos basados en la triangulación de Delaunay [George et al., 1991], [Carey, 1997], [George y Borouchaki, 1998], [Thompson et al., 1999] y [Frey y George, 2000]. Una vez que se ha discretizado la geometría del dominio, la malla debe adaptarse atendiendo a las singularidades de la solución numérica. Este proceso implica la introducción (refinamiento) o eliminación (desrefinamiento) de nodos de la malla actual. Los cambios pueden afectar a la malla actual de forma local o global, dependiendo del método de triangulación elegido. Diferentes estrategias de refinamiento han sido desarrolladas para triangulaciones en 2-D, y han sido generalizadas a 3-D.

Si se ha optado por un refinamiento que afecte localmente a la malla actual, cabe plantease otra cuestión: ¿mallas encajadas o no encajadas?. La respuesta en este caso no es tan clara. El uso de mallas encajadas tiene varias ventajas importantes. Se pueden conseguir familias de secuencias de mallas encajadas en un mínimo tiempo de CPU. Además, se puede aplicar más fácilmente el método multimalla para resolver el sistema de ecuaciones asociado al problema. Por otra parte, se puede controlar automáticamente la suavidad y la degeneración de la malla, y el mantenimiento de las superficies definidas en el dominio, en función de las características de la malla inicial. Si el dominio posee una geometría compleja, un buen modo de proceder es obtener la malla inicial empleando un generador de mallas no estructuradas y, posteriormente, aplicar una técnica de refinamiento y desrefinamiento local de mallas encajadas atendiendo a un indicador de error apropiado al problema. Además, si se trata de resolver un problema evolutivo, se puede aproximar automáticamente cualquier solución inicial definida en el dominio. Con la técnica de refinamiento y desrefinamiento se consigue un óptimo soporte de interpolación a trozos capaz de aproximar esta solución con la precisión deseada. En general, podría aplicarse esta técnica para cualquier función definida en el dominio de forma discreta o analítica.

Con estas ideas, anteriormente se desarrollaron técnicas adaptables en 2-D obteniendo buenos resultados en diferentes problemas estacionarios y evolutivos, por ejemplo [Ferragut et al., 1994], [Montenegro et al., 1997], [Winter et al., 1995]. En estos trabajos se utilizó una versión del algoritmo de refinamiento local 4-T de Rivara [Rivara, 1987]; todos los triángulos que deben ser refinados, atendiendo al indicador de error, se dividen en cuatro subtriángulos mediante la introducción de un nuevo nodo en los centros de sus lados y uniendo el nodo

introducido en el lado mayor con el vértice opuesto y los otros dos nuevos nodos (figura 1.1(a)). La elección particular del algoritmo de refinamiento es muy importante, puesto que el algoritmo de desrefinamiento puede entenderse como el inverso del algoritmo de refinamiento. El algoritmo de refinamiento 4-T de Rivara posee buenas propiedades en cuanto a la suavidad y degeneración de la malla. Además de esto, el número de posibilidades que aparecen en la relación entre un elemento padre y sus hijos es menor que con otros algoritmos de refinamiento en 2-D, tras asegurar la conformidad de la malla. Sería más complicado desarrollar un algoritmo de desrefinamiento, acoplado con el algoritmo de refinamiento local propuesto en [Bank et al., 1983]; todos los triángulos que deben ser refinados, atendiendo al indicador de error, se dividen en cuatro subtriángulos mediante la introducción de un nuevo nodo en los centros de sus lados y uniéndolos entre sí (figura 1.1(b)).



(a) Algoritmo 4-T de Rivara (b) Algoritmo de Bank

Figura 1.1: Divisiones del triángulo en 2-D

En 3-D, el problema es diferente. Aunque parezca paradógico, la extensión de un algoritmo adaptable que sea más simple que otro en 2-D, no tiene porqué ser también más simple en 3-D. Así, entre los algoritmos de refinamiento desarrollados en 3-D se puede mencionar los que se basan en la bisección del tetraedro [Arnold et al., 2001], [Rivara y Levin, 1992], [Plaza y Carey, 2000], y los que utilizan la subdivisión en 8-subtetraedros [Bornemann et al., 1993], [Liu y Joe, 1996], [Löhner y Baum, 1992]. En concreto, el algoritmo desarrollado en [Plaza y Carey, 2000] se puede entender como la generalización a 3-D del algoritmo 4-T de Rivara, que a su vez está basado en la bisección del triángulo por su lado mayor. El problema que se produce en esta extensión a 3-D es el gran número de casos posibles en los que puede quedar dividido un tetraedro, respetando las diferentes posibilidades de la división 4-T en sus cuatro caras, durante el proceso de conformidad de la malla. Sin embargo, los algoritmos analizados en [Bornemann et al., 1993], [Liu y Joe, 1996], [Löhner y Baum, 1992],

que a su vez generalizan a 3-D la partición en cuatro subtriángulos propuesta en [Bank et al., 1983], son más sencillos debido a que el número de particiones posibles de un tetraedro es mucho menor que en el caso de la generalización del algoritmo 4-T.

1.2. Justificación

Los algoritmos de refinamiento y desrefinamiento se ha convertido en herramientas fundamentales en el procesamiento de mallas. No sólo a la hora de ajustar la solución numérica en la aplicación de métodos de resolución de problemas, sino en el mismo proceso de generación de la malla inicial que ajuste el dominio pueden ser empleados satisfactoriamente. En definitiva, en cualquier proceso que implique un tratamiento de mallas será importante disponer de dichos algoritmos.

Por otro lado, y como ya se mencionó, en 3-D la complejidad de los algorimos es mayor que en 2-D. El hecho de disponer de algoritmos relativamente simples facilita, además de la programación, la aplicación de los mismos en diversos tipos de problemas.

En este trabajo se propone una implementación de unos algoritmos de refinamiento/desrefinamiento en 3-D en los que se han simplificado mucho los casos de conformidad, a costa de aumentar, en algunos casos, la introducción de nuevos elementos en la malla.

Tradicionalemente, el leguaje de programación FORTRAN ha sido empleado en el desarrollo de aplicaciones numéricas. Este lenguaje ha sufrido importantes revisiones (1977, 1990, 1995 y 2003), pero el lenguaje *C* aporta una serie de valores añadidos a tener en cuenta a la hora de definir y procesar estructuras de datos complejas. El *C++*, también denominado *C orientado a objetos* permite emplear las estructuras de datos de forma más natural, incluyendo en las propias estructuras los métodos que hacen uso de los datos, por lo que se ha empleado para el desarrollo.

1.3. Objetivos

Puesto que la calidad de la malla está asegurada en todos los casos de división que se han expuesto, el objetivo de este trabajo es la implementación de un

algoritmo de refinamiento/desrefinamiento para mallas de tetraedros basado en la subdivisión en 8 tetraedros. Se han tenido en cuenta los siguientes aspectos:

- Independencia del problema modelado. Los algoritmos van a operar sobre mallas, considerándolas una estructura no acoplada a un problema. El efecto del algoritmo sobre la malla vendrá dado por los indicadores de refinamiento o desrefinamiento aplicados sobre los elementos. Esto permitirá emplearlos en cualquier tipo de problema.

- Integración con módulos prevamiente desarrollados, por lo que las interfaces de comunicación deben ser amplias y versátiles. El desarrollo deberá poder ser enlazado (compilado) con otros módulo o bien comunicarse mediante estructuras de datos compartidas (en fichero o en memoria).

- Portabilidad entre sistemas, que viene garantizada en buena medida por la elección del *C++* como lenguaje de programación puesto que está disponible en prácticamente todos los sistemas. Aunque se han empleado librerías externas, estas están incluidas dentro del estándar de *C++*, habiendo realizado un desarrollo propio para aquellas rutinas que no figuran en los estándar internacionales.

- Escalabilidad que permita añadir funcionalidades, datos o rutinas de proceso sin alterar significativamente el desarrollo original.

1.4. Metodología

Para alcanzar los objetivos mencionados se ha realizado un desarrollo completo empleando el lenguaje *C++* en diferentes compiladores y sistemas. Ha sido probado en máquinas con *linux* y Windows usando compiladores de GNU desde la versión 3.0.

En el capítulo 2 se muestran una serie de herramientas empleadas para la implementación de los algoritmos, así como para la modelización de los elementos de la malla. Los modelos de datos diseñados para cada elemento recogen las particularidades de cada uno, si bien ha sido necesario elaborar una jerarquía compleja para mantener las relaciones de vecindad entre ellos necesarias para realizar las particiones de los mismos.

Para la sincronización de procesos concurrentes se estudiará la librería para procesamiento paralelo *POSIX*, que aunque no está presente en todos los sistemas, ha sido incluida mediante una clase que abstrae todas las llamadas y es fácilmente modificable (o incluso anulable) para permitir su uso es entornos donde no esté disponible el estándar *POSIX*.

En los capítulos 3 y 4 se presentan las implementaciones de los algoritmos de refinamiento y desrefinamiento respectivamente. Como se verá, no se emplea en ningún caso el sistema tradicional de numeración local de nodos, sino un etiquetado soportado por una serie de estructuras de datos que referencian los nuevos elementos creados a partir de otros.

El capítulo 5 está dedicado exclusivamente a aplicaciones del algoritmo. En todos los casos el objetivo es, dada una malla base, obtener una nueva mediante la adición/supresión de elementos que ajuste mejor la solución de un problema. En unos caso el objetivo es obtener una solución numérica de un problema mientras que en otro se trata de obtener una malla de la mejor calidad posible. Son casos diferentes en los que el uso del algoritmo ha supuesto un incremento en la precisión de la solución del problema.

Finalemente, en el capitulo 6 se establecen unas conclusiones y se plantean una serie de líneas futuras posteriores a este trabajo.

Capítulo 2

Desarrollo

2.1. Programación

Todo el desarrollo de la aplicación se ha realizado en lenguaje *C++*. Este lenguaje es un superconjunto del primitivo lenguaje *C*.

C es un lenguaje de programación creado en 1969 por Ken Thompson y Dennis M. Ritchie en los Laboratorios Bell. Es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix. No es un lenguaje de muy alto nivel, por lo que genera un código muy eficiente.

C es un lenguaje estandarizado, lo que permite la portabilidad entre sistemas. Aunque cada compilador ofrece sus propias extensiones, una aplicación desarrollada en *C* estándar podría trasladarse prácticamente a cualquier entorno. Fue en 1989 cuando se publicó la primera versión de lo que se conoce como *ANSI-C*, una serie de especificaciones del lenguaje que todos los compiladores deberían soportar. En esta definición se formalizaba el *C* original, estableciendo una serie de pautas que quedaban confusas en el lenguaje original.

En el año 1999 se publica un nuevo estándar por parte de la ISO, en la que se incluyen nuevos tipos de datos y ciertas características que lo hacían más parecido al *C++*. Este estándar no ha sido muy seguido por algunas compañías, que han preferido centrar sus esfuerzos en el *C++*.

C++ fue diseñado a mediados de los años 1980, por Bjarne Stroustrup, como extensión del lenguaje de programación *C*. Actualmente existe un estándar, denominado *ISO C++*, al que se han adherido la mayoría de los fabricantes de compiladores más modernos. *C++* está considerado por muchos como el lenguaje más potente, debido a que permite trabajar tanto a alto como a bajo

nivel. Sin embargo es a su vez uno de los que menos automatismos trae ya que obliga a hacerlo casi todo manualmente.

2.1.1. Tipos de datos y clases

Los tipos de datos nativos de *C* eran muy escasos, básicamente los que podía manejar el hardware de la máquina, por lo que debía ser el programador el que implementara la gestión de tipos más complejos. Puesto que, además, era un lenguaje orientado a la programación de sistemas, el soporte para cálculos numéricos dejaba mucho que desear.

En posteriores revisiones del lenguaje *C*, y ya con la aparición del *C++*, se dotó de nuevos tipos de datos más elaborados y de una potente capacidad de cálculo, lo que lo hacían perfectamente válido para aplicaciones numéricas.

En *C++* se pueden definir tipos de datos tan complejos como se deseen mediante el uso de clases. Una clase define un tipo de dato y qué operaciones de pueden realizar con él. Cada clase debe definir que zonas son públicas y/o privadas, de manera que se permita el acceso desde el exterior de la misma.

Las clases proveen de un método para realizar una abstracción de un ente que se quiera modelizar, dotándolo de propiedades y de métodos que manejen dichas propiedades. La clase en sí misma es una definición de como se va a modelar. Un objeto es una instancia de una clase, una ocurrencia de ésta, que tiene los atributos definidos por la clase, y sobre la que se puede ejecutar las operaciones definidas en ella.

Sobre las clases se puede realizar herencia: una clase que hereda de otra tendrá todos sus métodos y propiedades, además de los que se definan como propios. Esto permite generar jerarquías de clases y desarrollar modelos más rápidamente y mejor ajustados a lo que se pretende simular.

2.1.2. Plantillas

Con las plantillas se maneja el concepto de programación genérica. Permiten que una clase trabaje con tipos de datos abstractos, especificándose más adelante cuales son los que se quieren usar.

Gracias a las plantillas, no es necesario definir una clase específica para cada tipo de dato. Se puede generar una plantilla que no haga referencia a ningún tipo concreto y el compilador, en función del uso dado a la plantilla, generará

el código adecuado.

Se han usado ampliamente las plantillas en este trabajo. En la sección 2.2 se presenta un esquema que está totalmente basado en plantillas.

2.1.3. Punteros

Un puntero (en *C* y en *C++*) en una referencia de una zona de memoria. A través de la referencia podemos acceder al contenido de los datos apuntados.

En caso de tener que pasar datos complejos como parámetros de procedimientos/funciones, es mucho más eficiente pasar el puntero que dichos datos, puesto que un puntero simplemente ocupa el equivalente a un entero (una *palabra* de la CPU).

2.2. La *Standard Template Library*

La *Standard Template Library* [Stepanov y Lee, 1995] es una colección de estructuras de datos genéricas y algoritmos escritos en C++. No es la primera de este tipo de librerías. La mayor parte de los compiladores de C++ disponen de librerías de este tipo, y además existen de tipo comercial. El problema es que son incompatibles entre ellas.

Sin embargo, la STL ha sido adoptada por el comité ANSI para la estandarización del C++, lo que implica que está soportada como una extensión del lenguaje por todos los compiladores.

La STL proporciona una colección de estructuras de datos contenedoras y algoritmos genéricos que se pueden utilizar con éstas. Una estructura de datos se dice que es contenedora si puede contener instancias de otras estructuras de datos. De las diferentes disponibles se han utilizado las siguientes:

- Vectores (*vector <>*). Este contenedor se utiliza como depositario de los resultados finales de un refinamiento o desrefinamiento sobre la malla, permitiendo un acceso directo a cualquiera de los elementos que contiene mediante su índice numérico. El coste del acceso es de tipo $O(1)$. Las inserciones y extracciones de elementos en este contendedor, en teoría, también serían de coste $O(1)$, puesto que son capaces de adaptarse a la nueva dimensión adquiriendo o devolviendo memoria al sistema. Pero en la práctica, a medida que crecen las demandas de memoria de la aplicación,

un crecimiento del espacio inicialmente asignado provocaría el movimiento de todos los elementos del contenedor, pasando a tener coste $O(n)$ (muy alto cuando la operación se realiza con frecuencia). En cuanto a la devolución de pequeños trozos de memoria no utilizada genera una excesiva fragmentación de la memoria disponible. En los procesos en los que es necesario realizar frecuentes inserciones y extracciones se emplean listas enlazadas, usándose los vectores en el final del proceso. En general, cuando se conoce previamente el número de elementos que tendrá la estructura, y este número no va a tener cambios, los vectores son las estructuras más adecuadas.

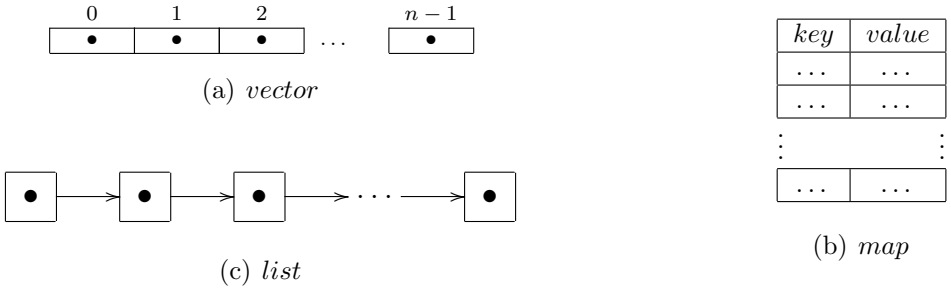
- Listas enlazadas (*list* <>). Las listas enlazadas presentan muchas ventajas en procesos de revisión de elementos sin un orden determinado. A medida que durante el proceso van surgiendo nuevos elementos a revisar van siendo añadidos a la lista, mientras que el proceso de revisión en sí los va eliminando. Estas frecuentes operaciones de inserción y extracción tienen tan solo coste $O(1)$. Otro uso de este contenedor se le da en una clase que añade funcionalidades para recorrer (iterar) sobre elementos mediante una copia de los mismos.

- Colas (*queue* <>). Implementan la política *FIFO* (*Fisrt-in*, *First-out*) de tratamiento de datos. Sólo se emplea en el control de elementos divididos en los procesos paralelos [3.4].

- Contenedor asociativo (*map* <>). Permite acceder a pares de elementos en forma de array asociativo. Se emplea para almacenar los parámetros del problema que son leídos en forma de fichero *.ini*.

- Iteradores (*iterators*). Esto no es un contenedor en sí mismo, sino que son operadores de iteración para recorrer eficazmente los elementos de un contenedor. Se emplean los asociados a vectores y listas.

Originalmente, en las primera implementaciones del algoritmo de refinamiento [González-Yuste et al., 2004b] se diseñó un contenedor propio llamado $Vector$. Este contenedor incluía gestión de memoria y poseía una característica: no habían elementos duplicados. En principio, este contenedor cumplía perfectamente con las necesidades de la programación, y ahorraba muchas verificaciones antes de añadir elementos al contenedor, pues éste ya se encargaba de evitar



(a) *vector*

(b) *map*

(c) *list*

Figura 2.1: Organización en memoria de los contenedores

duplicidades. Pero a la hora de establecer el rendimiento del sistema se comprobó que se ralentizaba considerablemente en todo lo relativo a este contenedor. Por el hecho de tener un almacenamiento tipo *array*, de fácil indexación, se perjudicaban las operaciones más frecuentes: inserciones y extracciones. En la Tabla 2.1 se puede ver los costes asociados a cada tipo de contenedor.

Estructura	**Inserción**	**Extracción**	**Acceso directo**
STL vector <>	$O(1) \rightarrow O(n)$	$O(1) \rightarrow$ fragmentación memoria	$O(1)$
STL list <>	$O(1)$	$O(1)$	$O(n)$
Vector	$O(n)$	$O(1) \rightarrow$ fragmentación memoria	$O(1)$

Tabla 2.1: Comparación de costes entre diferentes estructuras

Tras estudiar con detalle las estructuras de datos que hacían falta en cada una de los módulos de los programas, se determina que hacen falta dos contenedores básicos: los vectores y las listas. Los primeros porque es imprescindible ofrecer un acceso indexado a los elementos, más aún si se trata de módulos externos (generación de ficheros, importación de resultados, etc.). Las listas son las estructuras con mejor rendimiento en cuanto a inserciones y extracciones (en primera/última posición), y se comprobó que muchos recorridos y comprobaciones podían realizarse con listas.

Puesto que, como ya se mencionó, la mayoría de los compiladores de C++ traen una implementación de la *Standard Template Library* se optó por utilizar estos recursos, y no realizar una implementación propia. Se adaptó, por tanto, el desarrollo de los algoritmos a las especificaciones de la *STL*, y para ello hubo de hacerse unas modificaciones:

- La gestión de la memoria de traspasó del contenedor a los propios elementos. De esta forma se clarificaba el tratamiento de los nuevos elementos

y la destrucción de los viejos por parte de un elemento "padre". En la Sección 2.5 se detallará.

- Añadir control de recorrido para cada elemento. Para agilizar los recorridos frecuentes se ha empleado una comprobación simple acerca de si un elemento concreto ya ha sido procesado. Inicialmente se crea una lista de elementos a estudiar. Se van eliminando elementos a medida que se vayan procesando, y se añaden nuevos elementos a estudiar. En este esquema iterativo es frecuente que se pase por un elemento más de una vez, por lo que para no tener que estudiarlo dos veces hay que verificar que ya lo ha sido. Habría dos opciones: no añadirlo nuevamente a la lista o añadirlo y antes de estudiarlo verificar que ha sido procesado. En ambos caso se necesita una comprobación rápida. Los detalles de esta implementación se pueden ver en la Sección 2.5.5.

- Conversión de listas a vectores. En el proceso de obtención de una nueva malla a partir de la anterior, bien sea por refinamiento como por desrefinamiento, es necesario establecer tanto los elementos que permanecen invariantes como los que se añadirán por división (nuevos) o por supresión (se añade el predecesor). En estos caso es habitual tener que tratar varias veces el mismo elemento, lo cual no era problema con la estructura $Vector$, ya que por definición eliminaba duplicados. Con la *STL* se empleará la estructura *list* para añadir todos los elementos, aún duplicados. Como el destinatario debe ser una estructura *vector*, sin duplicados, es necesario filtrar los datos en *list*, y para ello se emplean métodos ya implementados en la *STL* para ordenar y suprimir duplicados adyacentes. El resultado (una lista sin duplicados) se convierte rápidamente en un *vector*. Este método tiene un coste global de $O(n \log_2 n)$ (por la ordenación, ya que la supresión de adyacentes es de $O(n)$) frente al $O(n^2)$ para una inserción con comprobación de no duplicados.

La *Standard Template Library* nos ha proporcionado unas herramientas para el desarrollo a través de sus estructuras de datos que han sido de muchísima utilidad. Al ser módulos libres de errores no ha habido que preocuparse por su implementación, y únicamente se ha tenido que adaptar el desarrollo inicial. El esfuerzo ha merecido la pena, pues el aumento de la velocidad ha sido más que notable (en torno al 70 %). El diseño inicial cumplió con las necesidades de

validar el algoritmo propuesto, pero no tenía calidad para un uso en aplicaciones.

2.3. *POSIX threads*

POSIX es el acrónimo de Portable Operating System Interface for UNIX. Es una familia de estándares de llamadas al sistema definida por el *IEEE* y especificados en la referencia *IEEE 1003* y a nivel internacional con la referencia *ISO/IEC-9945*. *POSIX* especifica interfaces de usuario y software con el Sistema Operativo que permiten la portabilidad de programas entre diferentes sistemas abiertos. Esta característica es la que ha decidido el empleo de este estándar para la implementación de ciertas funcionalidades del sistema.

Un *thread* (hilo) [Ceballos, 2003] es una secuencia de instrucciones que es ejecutada en paralelo junto a otras secuencias. Permite que, para un mismo proceso, diferentes *threads* puedan llevar a cabo tareas de forma concurrente. *POSIX* define llamadas al Sistema Operativo para la creación y eliminación de *threads*, así como para la sincronización de los mismos mediante diferentes herramientas: *mutex* y variables de condición.

2.3.1. *Threads*

Cuando un Sistema Operativo crea un proceso, éste puede verse como un hilo primario. A su vez, podrá crear otros hilos para que se ejecuten de forma concurrente. Los hilos de un proceso comparten el mismo espacio de memoria y los recursos asignados por el Sistema Operativo. Cada hilo posee su propio espacio de pila, contador de programa y registros.

La generación de múltiples hilos en un proceso permite el tratamiento en paralelo de tareas en una misma máquina con varias CPU. Al compartir todos los hilos el mismo espacio de memoria, la sincronización entre ellos es mucho más simple que entre los procesos que se ejecutan en diferentes máquinas.

Un hilo puede pasar por diferentes estados. En la figura 2.2 se puede ver un diagrama básico de estos estados.

Un hilo *Nuevo* es el que ha sido creado, pero aún no ha sido activado. Cuando se active pasa al estado *Preparado*. Sólo los hilos en este estado pueden tener CPU asignada para su ejecución. Cuando esto sucede pasan al estado de *En ejecución*. Durante la ejecución de un hilo se puede producir un bloqueo, lo que detendrá la ejecución del hilo. Esto puede venir motivado porque el propio

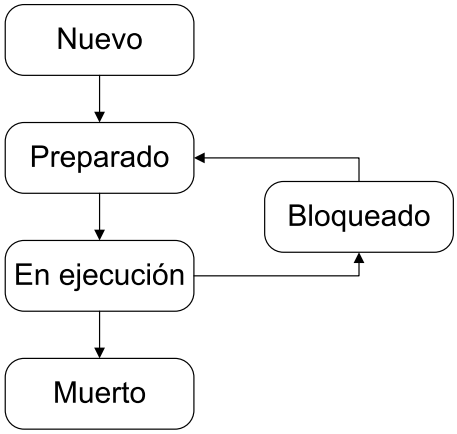


Figura 2.2: Diagrama básico de estados de un *thread*

hilo ha decidido suspender su ejecución (*Dormido*) o porque el hilo espera por una condición (*Esperando*): un objeto de sincronismo, un recurso de E/S, una condición, etc. Cuando lo adquiere volverá al estado de *Preparado*, quedando en cola para obtener tiempo de CPU. Finalmente, cuando un hilo concluye su tarea pasa al estado *Muerto*.

2.3.2. *Mutex* y Secciones críticas

Una *sección crítica* es un trozo de código cuya ejecución por parte de diferentes hilos podría dar resultados no deseados. El acceso a memoria compartida para escribir o, en general, a un recurso en modo escritura por parte de varios hilos de forma simultánea se puede considerar una sección crítica. Incluso accesos de lectura podrían ser secciones críticas si se realizan sobre un recurso adquirido por otro hilo sin que haya completado su escritura. Los resultados obtenidos podrían ser erróneos.

Las secciones críticas deberían ser ejecutadas sólo por un hilo cada vez. Es necesario disponer de mecanismos de sincronismo que bloqueen los hilos que pretendan acceder a recursos adquiridos por otros hilos. El mecanismo utilizado en este trabajo es el *mutex*.

El algoritmo 2.1 describe el trabajo de una sección crítica empleando objetos mutex como sincronizadores. Un mutex, acrónimo de *mutual exclusion* (mecanismo de exclusión mutua), es un objeto que sólo puede ser adquirido por un hilo cada vez. Si otro hilo intentara adquirir el mutex quedará bloqueado a la espera de que sea liberado. Los objetos mutex deben ser declarados a nivel glo-

bal, de manera que puedan ser accedidos por todos los hilos que ejecuten una determinada sección crítica.

Algoritmo 2.1 Algoritmo de ejecución de sección crítica con mutex

```
Adquirir mutex
    Sección crítica
Liberar mutex
```

En la figura 2.3 se puede ver como queda el diagrama de estados para un hilo cuando emplea mutex para ejecutar secciones críticas.

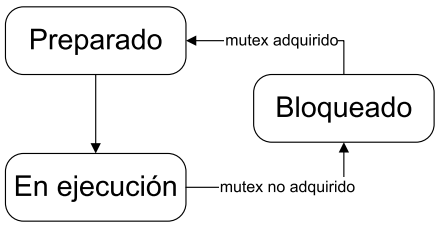


Figura 2.3: Diagrama de estados de un *thread* usando *mutex*

2.3.3. Variables de condición

Son otra herramienta de sincronismo entre hilos. En algunas ocasiones la sincronización se debe dar a lo largo del proceso y no al final del mismo. Los mutex no pueden resolver este problema de forma satisfactoria, y son las variables de condición las indicadas para estos casos. Un ejemplo típico es el problema del productor-consumidor: un proceso se encarga de generar datos continuamente, mientras que otro proceso se encarga de tratarlos. El proceso productor debe depositar los nuevos datos en alguna estructura, siempre con acceso exclusivo a la misma. El proceso consumidor extrae datos de la estructura para procesarlos.

Desde el punto de vista de los mutex el proceso productor sería sencillo: bloquear mutex, insertar dato, desbloquear mutex. Pero desde el punto de vista del consumidor la cosa no es óptima: debe esperar a que haya algún dato para poder trabajar. Para poder extraer un dato debe tener acceso exclusivo a la estructura bloqueando el mutex, pero en el supuesto de que no hubiera ninguno el proceso debe esperar, y debe liberar el mutex para que el productor pueda generar datos. Quedaría por tanto un bucle de espera que consume tiempo de CPU simplemente para comprobar que hay algún dato para procesar. Esto se puede ver en el algoritmo 2.2.

Algoritmo 2.2 Algoritmos productor-consumidor usando mutex

```
procedimiento PRODUCTOR
    Adquirir mutex
    Añadir dato
    Liberar mutex
fin procedimiento
```

```
procedimiento CONSUMIDOR
    Adquirir mutex
    mientras no haya datos
        Liberar mutex
        Adquirir mutex
    fin mientras
    Extraer dato
    Liberar mutex
fin procedimiento
```

Una variable de condición lleva un mutex asociado. Cuando un hilo bloquea una variable de condición bloquea a su vez el mutex asociado. Otro hilo que intentara adquirir la variable de condición quedaría en estado bloqueado, de forma similar a lo que sucede con el mutex. Pero las variables de condición permiten algo más. Un proceso consumidor puede ponerse en estado de espera, es decir, queda dormido hasta que un proceso productor le envíe una señal indicadora de que hay datos disponibles, liberando automáticamente la variable de condición. El proceso productor, una vez ha añadido el dato, envía la señal, y libera la variable de condición que previamente ha bloqueado. El proceso consumidor es despertado, y al reanudar su ejecución tendrá bloqueada la variable de condición. Los algoritmos 2.3 muestran la nueva forma de trabajar, en la que se soluciona la problemática de la espera activa.

Algoritmo 2.3 Algoritmos productor-consumidor usando variables de condición

```
procedimiento PRODUCTOR
    Bloquear variable
    Añadir dato
    Avisar al consumidor de que hay datos
    Liberar variable
fin procedimiento
```

```
procedimiento CONSUMIDOR
    Bloquear variable
    mientras no haya datos
        Esperar a que el productor avise liberando variable
    fin mientras
    Extraer dato
    Liberar variable
fin procedimiento
```

La variable de condición representa una cola de hilos en espera de que se reciba un aviso de que pueden continuar. Este aviso puede venir de dos formas:

- *signal*: Se despertará el primer hilo de la cola y pasará al estado de preparado.

- *broadcast*: Se despertarán todos los hilos en espera, siendo el planificador de CPU el que determine el siguiente hilo a ejecutar.

Otra forma de que un hilo sea despertado se da cuando se excede el tiempo de espera. Cuando un hilo se pone en espera se puede indicar un tiempo máximo. Si se supera ese tiempo el hilo despertará. Tendrá que comprobar si se activa por haber superado ese tiempo o por haber adquirido el mutex asociado a la variable de condición.

Con todo esto, en la figura 2.4, se puede ver el diagrama de estados para un hilo.

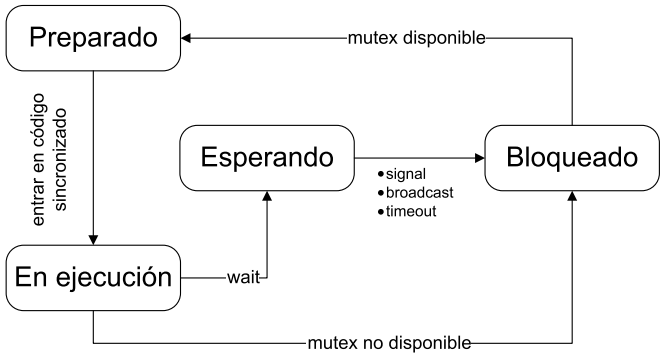


Figura 2.4: Diagrama de estados de un *thread* usando variables de condición

2.4. Modelo de objetos

Para realizar la implementación de los algoritmos se ha realizado una modelización de los objetos que intervienen en los mismos. Para cada uno se ha definido una clase, en muchos casos como herencia de otras, con los métodos y propiedades necesarios para cada uno.

La definición de las clases se ha agrupado en función de los objetos a modelar. Se puede detallar como:

- Objetos de la malla: definen los elementos que intervienen en las mallas (nodos, caras, aristas y tetraedros), además de definir clases auxiliares y comunes a todos ellos.

- Resolución de problemas: se han definido clases específicas para gestionar la resolución de problemas y el intercambio de información con otros programas.

- Estructuras de datos: son clases diseñadas para gestionar datos de manera eficiente, tales como iteradores, lectura/escritura de ficheros, hilos, mutex, etc.

2.4.1. Objetos de la malla

Puesto que la malla es el elemento principal de trabajo de esta implementación, se ha realizado un diseño de una jerarquía de clases orientadas al tratamiento de cada uno de los elementos que la componen. Aunque ya fue presentada en [González-Yuste et al., 2004b], ha sufrido modificaciones posteriores. En la figura 2.5 se puede ver el resultado, que se detallará a continuación.

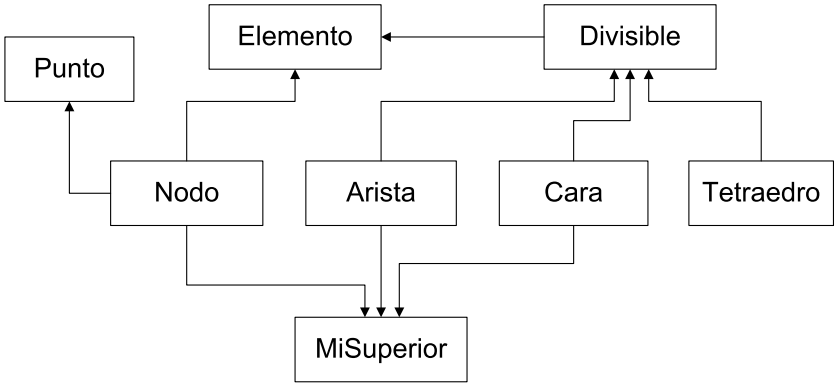


Figura 2.5: Jerarquía de clases de la malla

- **Punto**: define las propiedades de un punto en el espacio mediante sus coordenadas (x, y, z) y se incluyen algunos métodos para la suma y resta de coordenadas, producto vectorial de dos puntos y distancia al origen (módulo del vector asociado).

- **Elemento**: clase común para el resto de objetos con propiedades necesarias en todos: identificador único, número de referencia y nivel de profundidad en el que se encuentra en la malla. También tiene un indicador de *elemento destruido*, necesario para un proceso de limpieza en el desrefinamiento.

- **MiSuperior**: heredan de esta clase aquellas que sean elementos de otras (*nodos, aristas y caras*). Se mantienen referencias a los elementos superiores de cada una ($nodos \Rightarrow aristas$, $aristas \Rightarrow caras$ y $caras \Rightarrow tetraedros$).

- **Divisible**: esta clase mantiene referencias entre elementos padre y sus elementos hijos generados por una división. Heredarán de esta clase aquellas que sean susceptibles de poder dividirse (aristas, caras y tetraedros). Puesto que es heredada de *Elemento*, mantendrá sus propiedades en todas sus clases descendientes.

- **Nodo**: representa los nodos de una malla, heredando de *Elemento*, *Punto* y *MiSuperior<Arista>*. Mantiene un indicador de elemento desrefinable y los indicadores necesarios para almacenar una solución vectorial o escalar de un problema al que la malla esté asociada.

- **Arista**: hereda de *Divisible<Arista>* y *MiSuperior<Cara>* y modela las aristas de la malla. En estos elementos se almacena la marca de refinamiento para este proceso.

- **Cara**: hereda de *Divisible<Cara>* y *MiSuperior<Tetraedro>* y representa las caras que componen los tetraedros de la malla.

- **Tetraedro**: es la clase que modela los elementos que conforman la malla. Hereda únicamente de *Divisible<Tetraedro>*, y mantiene diferentes indicadores: si es transitorio, nuevo (en un proceso de refinamiento), refinable y los valores de una solución escalar o vectorial en la resolución de problemas.

Existen dos clases más que se han definido para ser usadas por algunas de las anteriores exclusivamente durante el proceso de división de los elementos:

- **OrientacionArista**: Mantiene referencias a los elementos internos de una arista. Incluye un operador para cambiar la orientación de los elementos desde el punto de vista de cualquier nodo.

- **OrientacionCara**: Mantiene referencias a los elementos internos de una cara. Al igual que la anterior, incluye un operador para poder variar el punto de vista de los elementos en función de los nodos a considerar.

El empleo de estas dos clases se verá en la parte dedicada a la implementación del algoritmo de refinamiento.

2.4.2. Clases para la resolución de problemas

Un grupo de clases han sido definidas para facilitar las interfaces entre diferentes módulos orientados a la resolución de problemas. Se han implementado accesos a ficheros para leer y/o escribir los resultados de un refinamiento/desrefinamiento en diferentes formatos, intercambio de estructuras de datos con módulos de suavizado de mallas o de resolución de ecuaciones y empaquetado de funciones en problemas específicos (problemas de viento).

La primera clase es la denominada *Malla*. Almacena referencias a objetos de las clases *Nodo*, *Arista*, *Cara* y *Tetraedro*. En esta clase se han implementado los procesos de refinamiento y desrefinamiento, módulos de compactación de listas y de eliminación de objetos suprimidos.

Independiente de la clase anterior, se ha definido también una pequeña jerarquía. La clase base es *Problema*. En esta clase se encuentra definida la malla inicial a tratar, que será pasada a un objeto de tipo *Malla* para aplicarles las operaciones necesarias. En la clase *Problema* hay interfaces para lanzar refinamientos/desrefinamientos, así como las rutinas necesarias de conversión de estructuras de datos entre diferentes aplicaciones (suavizado, resolución de ecuaciones, mallador).

Heredando de *Problema* está la clase *ProblemaFicheros*. Además de mantener todas las propiedades y métodos de la clase anterior, se añaden llamadas para la lectura y/o escritura de ficheros en múltiples formatos. También está programada la lectura de los ficheros de parámetros de la aplicación.

Para un tipo de problema específico se definió la clase *ProblemaViento*. Además de mantener los métodos de *ProblemaFicheros*, se han encapsulado en llamadas simples unas series de tareas muy repetitivas en la resolución de problemas de viento.

En caso de que hubiera otro tipo de problemas con una serie de tareas muy definidas se le podría definir una clase propia con una encapsulación de esas tareas y facilitar su llamada.

2.4.3. Clases que dan soporte a estructuras de datos

Para facilitar la programación de algunos módulos se han definido una serie de clases que encapsulan y automatizan una serie de procesos comunes.

2.4.3.1. Iterador

Además de los iteradores propios de la *Standar Template Library*, se ha definido un iterador para recorrer elementos de listas y vectores. La clase *VecIter<>* está basada en un tipo *list<>* de la STL, con la particularidad de que acepta listas y vectores como elementos de entrada en su constructor. Este iterador realiza una copia de los elementos (siempre punteros), y permite realizar un recorrido eliminando aquellos que se van procesando.

En definitiva, funciona a modo de cola, pero permite realizar operaciones sobre el contenedor original, puesto que se trabaja sobre una copia del mismo.

2.4.3.2. *Threads* y secciones críticas

Para el lanzamiento de *threads* se han definido dos clases, la *IThread* y la *PIThread*. La primera encapsula todas las llamadas a la librería POSIX para la creación y destrucción de hilos. La segunda, heredada de la primera, permite el paso de parámetros a la función punto de entrada del hilo.

Para el control de las secciones críticas de define la clase *Mutex*. Encapsula llamadas para el bloqueo y desbloqueo de *mutex* mediante las funciones de la librería POSIX. Del mismo modo, para trabajar con variables de condición se ha definido la clase *ICondVar*, que permite poner un proceso en espera de una variable de condición e informar mediante *signals* y *broadcast* la conclusión de una tarea.

Estas clases, en definitiva, realizan un encapsulamiento de las llamadas a las librerías POSIX. Se ha optado por esta implementación pues en caso de tener que lanzar el proceso en un entorno que no emplee el estándar POSIX sólo habría que modificar estas clases sin tocar el resto del módulos.

Hay una clase más, *IBuffer*, que se usa como una cola en la que se insertan y extraen elementos. La particularidad reside en que tanto el proceso de extracción como de inserción se consideran secciones críticas puesto que se pueden realizar de forma concurrente, y emplea un mutex y una variable de condición para su control.

2.4.3.3. Acceso a ficheros

Los ficheros son el medio de comunicación más empleado entre procesos. Se emplean para el paso de datos a módulos de suavizado/desenredo, como medio

de entrada de un mallador y como salida para la visualización de resultados. Las operaciones de lectura/escritura son muy frecuentes, y los formatos de los ficheros no difieren en gran medida entre los diferentes módulos. Una serie de clases se encargan de realizar estas tareas.

La clase *Iifstream*, heredada de ifstream (el estándar de entrada en C++), se sobrecarga para saltarse las líneas de comentarios y mantener el puntero de lectura siempre posicionado en el siguiente carácter del flujo de entrada. Esta clase es empleada en todos los accesos de lectura a fichero.

La clase *RegistroFichero* define un tipo de registro genérico, configurable según el tipo de entrada a realizar. La clase *FicheroDeRegistro* permite leer un fichero con un tipo de registro definido por la clase anterior y cargar los datos en una lista equivalente. Asimismo, se puede volcar en un fichero el contenido de una lista del tipo de registro adecuada.

Finalmente, la clase *Fichero* se emplea en el caso particular del uso de ficheros como datos de entrada para la creación de una malla, bien por cuatro ficheros como datos de entrada (datos de nodos, aristas, caras y tetraedros) o tan solo por dos (nodos y tetraedros por nodos).

2.4.4. Agregación y uso

Además de la jerarquía de clases, se han empleado otras relaciones entre las clases: la agregación y el uso.

La agregación sigue el predicado *es parte de*. La agregación se empleará en casos en que una clase sea una parte o un todo de otra. Así, en las definiciones mencionadas se puede decir que Nodo *es parte de* Arista, Arista *es parte de* Cara y Cara *es parte de* Tetraedro. De la misma manera Nodo, Arista, Cara y Tetraedro *son parte de* Malla.

Por otra parte, la relación de uso implica una dependencia de utilidad: una clase es usada por otra como elemento de ayuda, como la clase *Punto*, que es usada por *Tetraedro* para mantener los valores (x, y, z) de una solución a un problema de viento en ese elemento.

En la figura 2.6 puede verse un esquema completo de las relaciones jerárquicas, de agregación y de uso entre las diferentes clases.

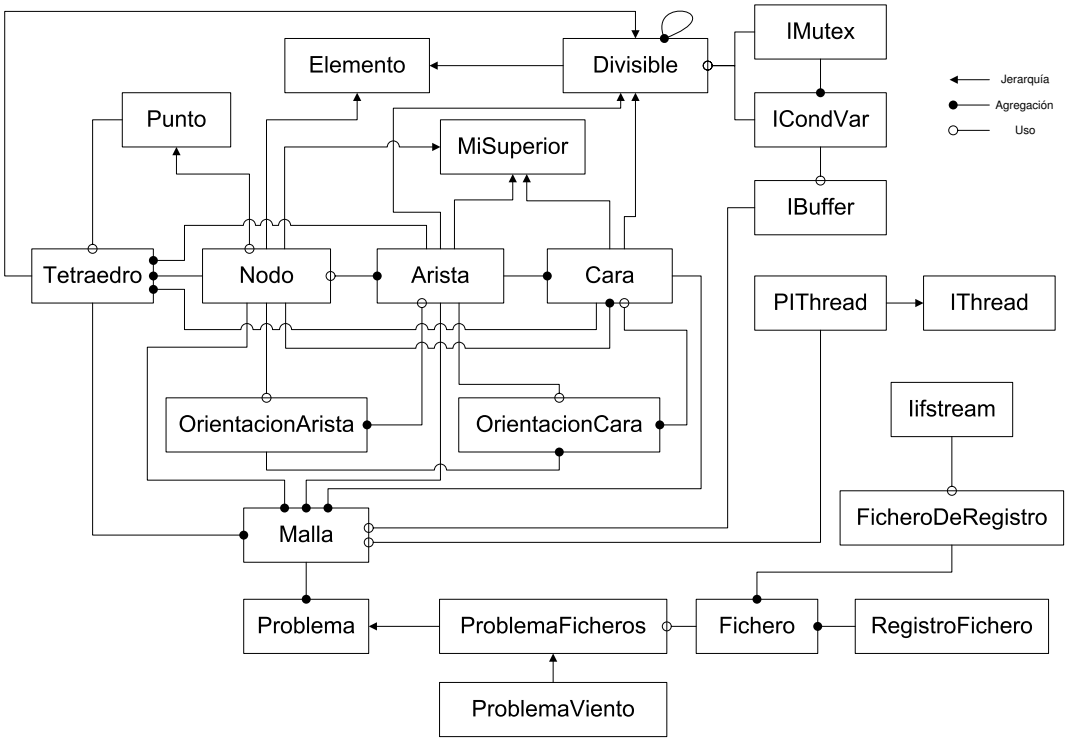


Figura 2.6: Jerarquía/agregación/uso de clases

2.5. Estructuras de datos

Como ya se comentó en la sección 2.2, acerca de la *Standard Template Library*, se han empleado básicamente, dos tipo de estructuras contenedoras de datos: las listas y los vectores. En la mencionada sección se ha realizado un análisis de coste, así como se ha comentado las circunstancias ideales para el uso de cada tipo de contenedor. En la implementación del algoritmo de este trabajo se han utilizado en función de las necesidades de cada caso.

Cabe destacar que de cada elemento de la malla sólo hay una instancia en la memoria. De resto, todo son referencias a dicha instancia. En las secciones 2.5.1 y 2.5.4 se detalla en dónde se almacena cada objeto.

2.5.1. Carga inicial de una malla

La primera rutina de la aplicación consiste en generar en memoria la estructura de una malla. Para ello se emplearán los datos de algunos ficheros de entrada. Un preproceso de los ficheros va a dar el número de elementos de ca-

da tipo (nodos, aristas, caras y tetraedros). Puesto que la malla inicial es fija (no cambia), se ha optado por usar una estructura vector para almacenar la configuración original.

Hay que destacar que lo que se almacenan son instancias de los objetos, es decir, objetos originales creados mediante sus constructores, no referencias a los mismos. La clase *Problema* será la encargada de mantener esta información mediante contenedores de tipo *vector<>*. Una vez la malla esté formada en memoria, se invocará a la clase *Malla* con una relación de referencias de cada tipo de elemento.

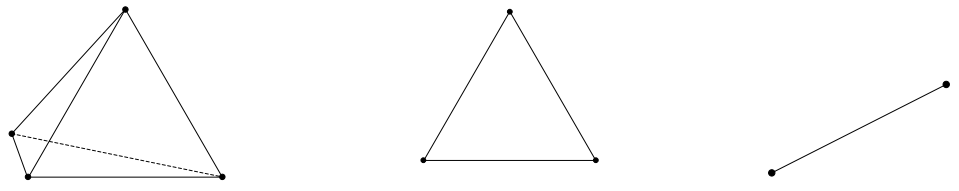
Si en la clase *Problema* se almacenan los objetos mismos, en la clase *Malla* se tienen las referencias a los mencionados objetos. La intención es que en *Malla* esté las referencias de los elementos que conforman la malla de trabajo en cada momento, inicialmente con los que la clase *Problema* le indica y alterando estas listas añadiendo y/o eliminando elementos cuando haya un proceso de refinamiento/desrefinamiento.

Puesto que en *Malla* va a haber una configuración estable, tanto al inicio como al final del proceso, se han empleado también contenedores tipo *vector* para mantener la información.

2.5.2. Relación entre elementos

Todos los elementos de la malla están relacionados entre ellos mediante dos predicados: "*está formado por*" y "*forma parte de*".

El primero de ellos implica una relación entre elementos de forma estática: un tetraedro (figura 2.7(a)) está formado por cuatro caras, seis arista y cuatro nodos, una cara (figura 2.7(b)) por tres aristas y tres nodos, y una arista (figura 2.7(c)) por dos nodos. Al ser una relación fija en número de elementos, se puede usar perfectamente el contenedor *vector<>* para almacenar las referencias a otros objetos.



(a) Tetraedro (b) Cara (c) Arista

Figura 2.7: Composición de los elementos de una malla

El segundo predicado establece qué elementos dan lugar a otros superiores: nodos que forman aristas, nodos y aristas que forman caras y nodos, aristas y caras que forman tetraedros. Puesto que no es posible saber a priori, por ejemplo, cuantas aristas están formadas por un determinado nodo, se emplean para estas relaciones contenedores tipo *list*<>, almacenando las referencias a los objetos de nivel superior.

2.5.3. Operadores de conjuntos

A partir de los predicados mencionados en la sección 2.5.2, se han definido una serie de conjuntos propios de cada elemento. Al haber una estructura jerarquizada en la relación, sólo se tienen almacenados, de forma directa, referencias a los elementos que componen a uno determinado (número fijo de elementos) y referencias a los elementos de nivel inmediatamente superior. En la tabla 2.2 se muestran estos conjuntos básicos.

(a) Tetraedro

Conjunto		Contenedor
C_t	Caras del tetraedro	*vector*
A_t	Aristas del tetraedro	*vector*
N_t	Nodos del tetraedro	*vector*

(b) Cara

Conjunto		Contenedor
Ω_c	Tetraedros formados por la cara	*list*
A_c	Aristas de la cara	*vector*
N_c	Nodos de la cara	*vector*

(c) Arista

Conjunto		Contenedor
X_a	Caras formadas por la arista	*list*
N_a	Nodos de la arista	*vector*

(d) Nodo

Conjunto		Contenedor
Λ_n	Aristas formadas por el nodo	*list*

Tabla 2.2: Conjuntos básicos de los elementos de una malla

Todos estos conjuntos son generados en los procesos de composición de cada elemento, a medida que se generan nuevos tetraedros, caras, aristas y nodos. Pero en los procesos de refinamiento/desrefinamiento es necesario acceder a elementos vecinos y grupos de elementos de niveles superiores que comparten uno dado. Definimos los vecinos de un elemento dado como todos aquellos que comparten cualquier otro elemento que lo forma. Ha sido necesario establecer unos operadores de conjuntos para recuperar toda esta información.

En el caso de los operadores (tabla 2.3), los contenedores empleados son de tipo *list*. La forma de implementar la obtención de alguno de los conjuntos de

(a) Nodo

Operador		
$X_n = \bigcup_j X_{a_j}$	$a_j \in \Lambda_n$	Caras desde nodo
$\Omega_n = \bigcup_j \Omega_{c_j}$	$c_j \in X_n$	Tetraedros desde nodo

(b) Arista

Operador		
$\upsilon_{a\times n} = \bigcup_j \Lambda_{n_j}$	$n_j \in N_a$	Vecinos por nodos
$\Omega_a = \bigcup_j \Omega_{c_j}$	$c_j \in X_a$	Tetr. desde arista

(c) Cara

Operador		
$\upsilon_{c\times n} = \bigcup_j X_{n_j}$	$n_j \in N_c$	Vecinos por nodos
$\upsilon_{c\times a} = \bigcup_j X_{a_j}$	$a_j \in C_a$	Vecinos por aristas

(d) Tetraedro

Operador		
$\upsilon_{t\times n} = \bigcup_j \Omega_{n_j}$	$n_j \in N_t$	Vecinos por nodos
$\upsilon_{t\times a} = \bigcup_j \Omega_{a_j}$	$a_j \in A_t$	Vecinos por aristas
$\upsilon_{t\times c} = \bigcup_j \Omega_{c_j}$	$c_j \in C_t$	Vecinos por caras

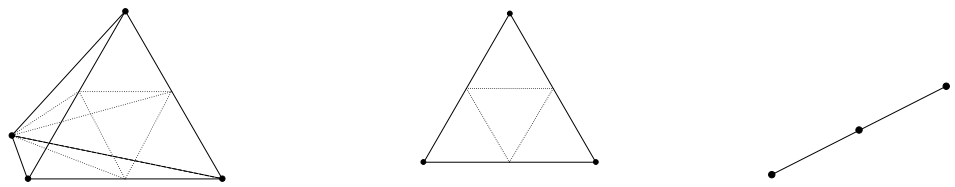
Tabla 2.3: Operadores de conjuntos

resultados de vecindad consiste en la adición de elementos en un contenedor y la posterior eliminación de los duplicados.

2.5.4. Generación/Eliminación de elementos

Durante un proceso de refinamiento/desrefinamiento se produce la eliminación y creación de nuevos elementos. Todos estos cambios tendrán reflejo en los contenedores de *Malla*.

Para que un elemento sea dividido siempre es necesario introducir otros elementos que realicen la partición. En las aristas, habrá que introducir un nodo en el medio para obtener dos nuevas aristas. En las caras siempre habrá aristas interiores que provoquen la división. Y en el caso de los tetraedros, será necesario introducir aristas y caras en su interior. En la figura 2.8 hay unos caso de división.



(a) Tetraedro (b) Cara (c) Arista

Figura 2.8: División de los elementos de una malla

Para mantener una adecuada organización en la gestión de la memoria, los

nuevos elementos son creados por el elemento que va a ser dividido. Internamente almacenará los objetos (no referencias) necesarios para su división y los objetos "hijo" en los que el elemento será dividido.

Esta configuración, en la que cada elemento almacena sus objetos divididos, ha simplificado el desarrollo, puesto que queda perfectamente localizado el lugar en el que reside cada nuevo objeto de la malla. En el diseño original los nuevos elementos era dependientes de la clase *Problema*, pero los procesos de eliminación provocaban bastantes incongruencias en la estructura de la malla. La estructura jerárquica que finalmente se utiliza ha resuelto todos los problemas.

Al finalizar los procesos de refinamiento/desrefinamiento se deben actualizar las referencias de la clase *Malla*, quitando las de los objetos que han sido eliminados o que han sido divididos y añadiendo la de los elementos generados.

2.5.5. Iteraciones sobre elementos

Una de las operaciones más frecuentes que se han realizado en el desarrollo de algoritmos son los recorridos por los elementos de cualquier contenedor. Se usan, básicamente, un contenedor *vector* y otro *list*. La *Standard Template Library* proporciona unos métodos para recorrer ambos contenedores con facilidad.

Aparte de los ya mencionados, hay que destacar dos operaciones que se realizan habitualmente. Se ha definido una clase *VecIter* como un caso particular de iteración. Admite cualquier contenedor como flujo de entrada, y su principal característica es que realiza una copia de las referencias del contenedor. Esto permite recorrer los elementos del contenedor (a través del *VecIter*) mientras se pueden hacer modificaciones sobre el contenedor original, aunque su principal virtud es servir de soporte a un operador de conjuntos. Los operadores podían estar en contenedores *list* o *vector*. Al poder recibir *VecIter* de ambos, permite desarrollos sin necesidad de detallar cada caso un recorrido específico para el contenedor.

El segundo recorrido consiste en realizar un procesamiento FIFO (colas) de los datos de una estructura. Este proceso es bastante usado, y mediante un bucle se van extrayendo y procesando los elementos de la cola. En algún caso se tendrá que añadir nuevos elementos a procesar en la cola, con lo que el bucle deberá continuar hasta que se complete el proceso. Se detalla en el algoritmo 2.4.

Algoritmo 2.4 Algoritmo de recorrido con procesamiento FIFO

```
procedimiento PROCESAR(datos-iniciales)
    lista= datos-iniciales
    mientras lista ≠ ∅                                  ▷ lista no vacía
        e = primer elemento de la lista
        eliminar primer elemento de la lista            ▷ e ←pop lista
        si e procesado entonces
            continuar bucle mientras
        fin si
        procesar elemento e
        si condición de procesamiento de e entonces
            Añadir a lista los elementos necesarios     ▷ lista ←push elementos
        fin si
    fin mientras
fin procedimiento
```

Capítulo 3

Algoritmo de Refinamiento

3.1. Presentación

El algoritmo implementado se basa en la subdivisión del tetraedro en ocho subtetraedros, ya presentado en [Löhner y Baum, 1992]. Esta implementación ya ha sido publicada en [González-Yuste et al., 2004b].

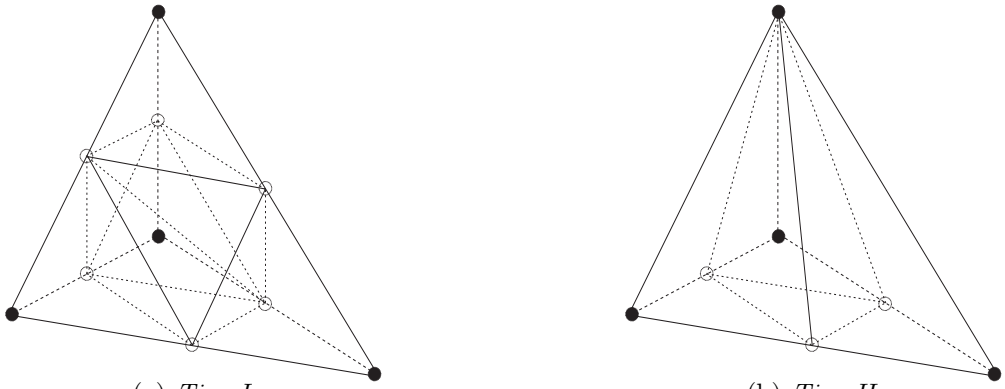
Partiendo de un mallado inicial del dominio M_0 formada por un conjunto de n_0 tetraedros $\tau_1^0, \tau_2^0, \ldots, \tau_{n_0}^0$, el objetivo es construir una secuencia de m niveles de mallas encajadas $T = \{M_0 < M_1 < M_2 < ... < M_m\}$, tal que el nivel M_{j+1} se obtiene mediante un refinamiento local del nivel anterior M_j. Cada elemento $\tau_i^j \in M_j$ tendrá asociado un indicador de error η_i^j, y será refinado si:

$$\eta_i^j \geq \gamma \eta_{\text{máx}}^j \tag{3.1}$$

η_{max}^j es el valor máximo del indicador de error de los elemenos de M_j. γ es el parámetro de refinamiento, donde $\gamma \in [0, 1]$. Para $\gamma = 0$ tendremos un refinamiento global, y con $\gamma = 1$ se realizará un refinamiento sólo en los elementos con el máximo valor del indicador de error.

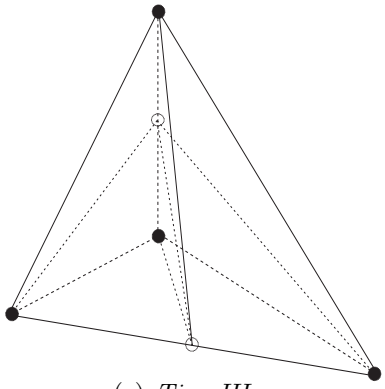
Desde un punto de vista constructivo, se plantea inicialmente la obtención de M_1 partiendo de la malla base M_0, atendiendo a las siguientes consideraciones:

1. *Subdivisión en 8-subtetraedros.* Decimos que $\tau_i^0 \in M_0$ es un tetraedro de *tipo I* si para su indicador de error η_i^0 se verifica la condición expuesta en la ecuación 3.1. Este conjunto de tetraedros serán posteriormente subdivididos en 8 subtetraedros según la figura 3.1(a): se introducen 6 nuevos nodos en el punto medio de sus aristas y se subdividen cada una de sus

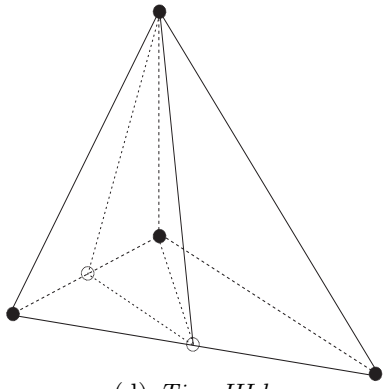


(a) *Tipo I*

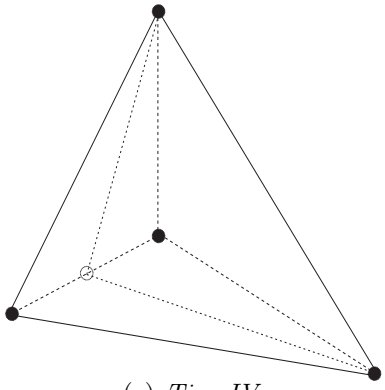
(b) *Tipo II*



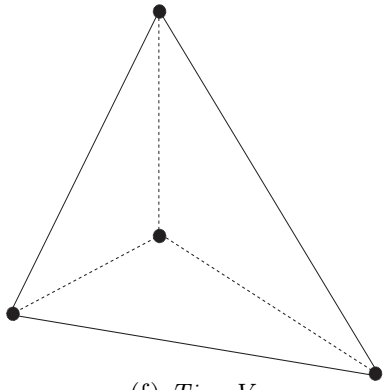
(c) *Tipo III.a*



(d) *Tipo III.b*



(e) *Tipo IV*



(f) *Tipo V*

Figura 3.1: Clasificación de las subdivisiones de un tetraedro en función de los nuevos nodos indicados con círculo blanco.

cuatro caras en cuatro subtriángulos. Cuatro subtetraedros quedan determinados a partir de los cuatro vértices de τ_i^0 y las nuevas aristas, y los otros cuatro subtetraedros se obtienen al unir dos vértices opuestos del octaedro que resulta en el interior de τ_i^0.

Una vez que se ha definido el tipo de partición de los tetraedros *tipo I*, de cara a asegurar la conformidad de la malla, nos podemos encontrar con tetraedros vecinos que pueden tener 6, 5,..., 1 ó 0 nuevos nodos introducidos en sus aristas. Analizamos a continuación cada uno de estos casos.

2. *Tetraedros con 6 nuevos nodos.* Aquellos tetraedros que por razones de conformidad tengan marcadas sus 6 aristas pasan automáticamente al conjunto de tetraedros *tipo I.*

3. *Tetraedros con 5 nuevos nodos.* Aquellos tetraedros que tengan 5 aristas marcadas pasan también al conjunto de tetraedros *tipo I.* Previamente, habrá que marcar la arista en la que no habría sido introducido ningún nuevo nodo.

4. *Tetraedros con 4 nuevos nodos.* En este caso, se marcan las dos aristas restantes y pasa a ser considerado de *tipo I.*

5. *Tetraedros con 3 nuevos nodos.* En este caso, hay que distinguir dos situaciones:

   a) Si las correspondientes 3 aristas marcadas no están sobre la misma cara, entonces se marcan las 3 restantes y el tetraedro se introduce en el conjunto de tetraedros *tipo I.*

   En los siguientes casos, ya no marcaremos ninguna nueva arista, lo cual implica que no se introducirá ningún nuevo nodo en el tetraedro que se pretende hacer conforme. Se procederá a subdividirlos de la forma que se indica a continuación, creando subtetraedros que llamaremos *transitorios*, ya que podrán desaparecer en posteriores etapas de refinamiento para asegurar que la malla no degenere.

   b) Si las 3 aristas marcadas están sobre la misma cara del tetraedro, entonces se crearán 4-subtetraedros transitorios como se muestra en la figura 3.1(b); se definen nuevas aristas uniendo entre sí los tres nuevos nodos y conectando estos con el vértice opuesto a la cara que los contiene. Los tetraedros de M_0 con estas características se englobarán en el conjunto de tetraedros de *tipo II.*

6. *Tetraedros con 2 nuevos nodos.* También distinguiremos dos situaciones:

a) Si las dos aristas marcadas no están sobre la misma cara, entonces se construirán 4-subtetraedros transitorios como se presenta en la figura 3.1(c), definidos a partir de las aristas que conectan los dos nuevos nodos y a estos con los vértices opuestos de las dos caras que los contienen. Los tetraedros que se encuentren en esta situación se denominan de *tipo III.a*.

b) Si las dos aristas marcadas están sobre las misma cara, entonces se crearán 3-subtetraedros transitorios según se expone en la figura 3.1(d); se divide en tres subtriángulos la cara definida por las dos aristas marcadas, conectando el nuevo nodo situado en la arista mayor de estas dos con el vértice opuesto y con el otro nuevo nodo, tal que estos tres subtriángulos y el vértice opuesto a la cara que los contiene definen los tres nuevos subtetraedros. Se destaca que de las dos posibles elecciones, se toma como referencia la mayor arista marcada para aprovechar en algunos casos las propiedades de la bisección por el lado mayor. Los tetraedros que se encuentren en esta situación se denominan de *tipo III.b*.

7. *Tetraedros con 1 nuevo nodo*. Se crearán dos subtetraedros transitorios según la figura 3.1(e); se une el nuevo nodo con los otros dos que no pertenecen a la correspondiente arista marcada. Este conjunto de tetraedros se denomina de *tipo IV*.

8. *Tetraedros con 0 nuevos nodos*. Estos tetraedros de M_0 no se dividen y serán heredados a la malla refinada M_1. Los denominaremos de *tipo V* y se representan en la figura 3.1(f).

Este proceso de clasificación de los tetraedros de M_0 se realiza simplemente marcando sus aristas. La conformidad de la malla se va asegurando por vecindad entre los tetraedros que contienen una nueva arista marcada, mediante un recorrido que comienza por los tetraedros de *tipo I* definidos en el apartado 1. Una vez se ha completado el proceso de marcado podrá comenzar el proceso de división y generación de nuevos elementos.

En general, cuando queremos refinar el nivel M_j en el que ya existen tetraedros *transitorios* se procederá de igual forma que en el paso de M_0 a M_1, con las siguientes variantes:

1. Si un tetraedro *transitorio* es marcado como refinable (Tipo I), se procederá a la eliminación de la división de su "padre" y éste será marcado como refinable (Tipo I).

2. Si una arista de un tetraedro *transitorio* es marcada por conformidad, se pueden dar dos casos:

   a) Si la arista está compartida con su tetraedro "padre", la división de este tetraedro "padre" será eliminada y se tendrá que estudiar en qué nuevo tipo incorporarlo.

   b) Si la arista no está compartida con su tetraedro "padre", se procederá de igual manera que el punto 1.

3.1.1. División en 8 tetraedros

Como ya se mencionó, esta división de tetraedros fue presentada por [Löhner y Baum, 1992]. La división de las caras del tetraedro en cuatro subtriángulos fue propuesta por [Bank et al., 1983] (figura 3.2).

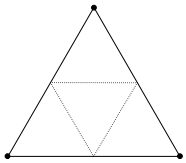


Figura 3.2: División de la cara en 4 subtriángulos

De la división de las caras de un tetraedro en cuatro subtriángulos cada una, surge un octaedro interior al tetraedro que deberá ser dividido en cuatro tetraedros mediante una arista que conecte dos puntos opuestos. Para esta división se presentan tres posibilidades que pueden verse en la figura 3.3:

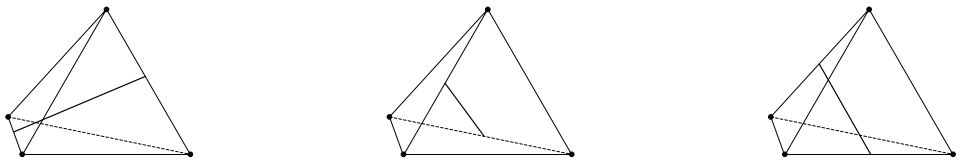


Figura 3.3: Posibles divisiones del octaedro interior

De las tres posibilidades, [Bornemann et al., 1993] y [Löhner y Baum, 1992] escogen unir los vértices más cercanos en cada caso. Por otra parte, [Liu y Joe,

1996] se basa en transformaciones afines a un tetraedro de referencia asegurando la calidad de los tetraedros resultantes. [Bornemann et al., 1993] afirma que con ambas estrategias obtiene resultados comparables en sus experimentos numéricos, y [Löhner y Baum, 1992] aseguran que la elección de unir los vértices más cercanos produce el menor número de tetraedros distorsionados en la malla refinada.

Por supuesto, siempre se podría realizar un análisis de la calidad de los tetraedros resultantes en cualquiera de las opciones, lo que supondría tener que calcular 3×4 cómputos de calidad por cada tetraedro que fuera a ser dividido. Debido a que el aumento en el coste computacional puede ser significativo, se ha optado por implementar la opción de unir los vértices más cercanos.

3.1.2. Propagación del algoritmo

En el caso de que un tetraedro reciba cinco, cuatro o tres marcas (en distinta cara), va a ser pasado a tetraedros de *Tipo I*, marcando las aristas que no lo estuvieran. Esto produce un efecto de propagación en el área de refinado a tetraedros que por el indicador de error no eran, inicialmente, de *Tipo I*.

En otros trabajos ([Gross y Reusken, 2005]) se define un conjunto completo de métodos de división. En concreto se habla de 64 modos de dividir el tetraedro. Por supuesto, esta opción elimina completamente el efecto "dominó" de la propagación, pero complica los procesos de división.

El hecho de contar con un conjunto reducido de formas de particionar un tetraedro aligera significativamente la carga del módulo de división. Por otro lado, en el caso de tetraedros con cinco o cuatro marcas, solo se tendrán que añadir uno o dos nuevos nodos sobre un total de seis, una proporción menor que la presentada en el algoritmo 4T de Rivara ([Rivara, 1987]) para dos dimensiones, en la que la probabilidad de tener el nodo sobre el lado mayor es de 1/3. Este efecto se acentúa en la generalización a 3D del algoritmo ([Plaza y Carey, 2000]).

En el caso de tener que introducir 3 nuevos nodos, se puede hacer la consideración antes mencionada si se le compara con algoritmos basados en la bisección por el lado mayor ([Arnold et al., 2001; Rivara y Levin, 1992]).

3.2. Implementación

La implementación del algoritmo se ha dividido en tres fases bien diferenciadas:

- Marcado
- División
- Compactación

Debido al uso de tetraedros transitorios, se hace necesario realizar un cierto número de recorridos sobre la malla aplicando las tres fases. En el primer recorrido se van a procesar todos los tetraedros marcados por el indicador de error, así como los transitorios marcados por conformidad. Pero hay un caso en el algoritmo que requiere de un segundo recorrido, puesto que un tetraedro puede verse sometido a dos procesos de división: cuando un tetraedro transitorio recibe una marca por conformidad en una de sus aristas propias (no compartida con el padre), el padre será dividido como *Tipo I*, y uno de sus hijos (el que tiene la arista marcada) volverá a ser dividido (figura 3.4).

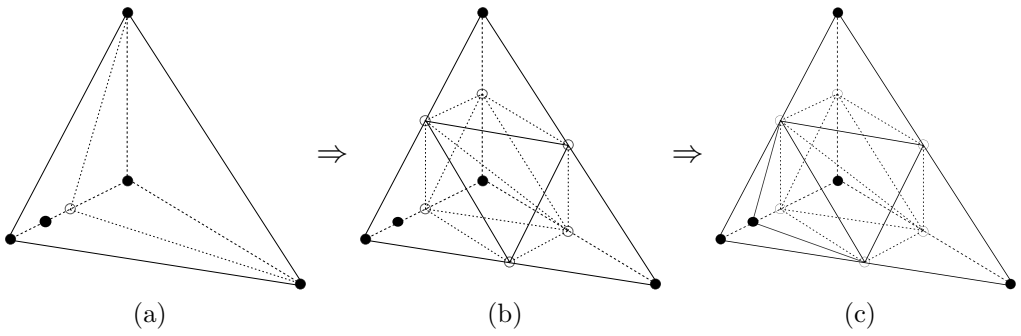


Figura 3.4: Secuencia de dos divisiones consecutivas

Puesto que ningún tetraedro puede sufrir más de dos divisiones en una etapa, serían necesarios, como máximo, dos recorridos sobre la malla. En el algoritmo 3.1 puede verse un esquema general del proceso de refinamiento para una malla M_j cualquiera.

Finalmente, hay que destacar que no se contempla el proceso de marcado de tetraedros para refinar por el indicador de error. La malla de entrada al proceso debe haberse revisado para tener señalados dichos tetraedros. Esto permite mayor versatilidad en el esquema general, puesto que permite realizar refinamientos locales atendiendo a otros criterios.

Algoritmo 3.1 Esquema general del proceso de refinamiento

```
procedimiento REFINAR(M_j)
    T_0 = M_j
    para k=1 hasta 2
        T_k = T_{k-1}
        MARCAR_MALLA(T_k)
        DIVIDIR_MALLA(T_k)
        COMPACTAR_MALLA(T_k)
    fin para
    M_{j+1} = T_2
fin procedimiento
```

3.3. Proceso de Marcado

Dentro del proceso general del refinamiento, la primera tarea consistirá en establecer qué elementos deberán ser divididos. El proceso de marcado contemplará el esquema del algoritmo de refinamiento en cuanto a propagación de marcas en aristas. En general, este proceso va a constar de dos subprocesos que realizarán recorridos sobre los tetraedros de la malla (algoritmo 3.2).

Algoritmo 3.2 Esquema del proceso de marcado

```
1: procedimiento MARCAR_MALLA(T_k)
2:     repetir
3:         ESTUDIAR_LISTA(T_k)
4:         hay_transitorios = ESTUDIAR_TRANSITORIOS(T_k)
5:     hasta (not hay_transitorios)
6: fin procedimiento
```

3.3.1. Clasificación de tetraedros

El primer recorrido realizará un estudio para determinar qué tetraedros son de *Tipo I*. En el algoritmo 3.3 puede verse un esquema del proceso. Se emplean los operadores definidos en la tabla 2.2. Por cuestiones de claridad se han añadido dos operadores, A^*_τ y A'_τ, que representan, respectivamente, las aristas marcadas y no marcadas de un tetraedro τ.

Dentro de este subproceso se contemplarán qué tetraedros son *Tipo I* debido al indicador de error y cuáles serán pasados a *Tipo I* debido a que han alcanzado en sus aristas el número de marcas necesarias (6, 5, 4 ó 3 marcas en distinta cara). En caso de encontrar un tetraedro que cumpla alguna de estas condicio-

nes, se le marcarán sus aristas no marcadas, y todos los tetraedros vecinos por estas aristas serán incluidos en la lista de tetraedros a estudiar. Si durante este bucle se encuentra un tetraedro transitorio, no se realizaría esta operación, y se dejaría para el segundo subproceso.

Algoritmo 3.3 Clasificación de tetraedros

```
1: procedimiento ESTUDIAR_LISTA(T)
2:     lista_de_tetraedros = τ ∈ T
3:     hay_transitorios = true
4:     mientras lista_de_tetraedros ≠ ∅
5:         τ ←pop lista_de_tetraedros
6:         si τ es transitorio or A*_τ = 6 entonces
7:             continuar bucle "mientras"
8:         fin si
9:         si τ refinable or A*_τ ∈ [5, 4, 3_{≠cara}] entonces
10:            para α_i ∈ A'_τ
11:                Marcar(α_i)
12:                lista_de_tetraedros ←push Ω_{α_i}
13:            fin para
14:        fin si
15:    fin mientras
16: fin procedimiento
```

3.3.2. Estudio de transitorios

En el segundo subproceso (algoritmo 3.4) se van a estudiar los tetraedros transitorios. Una vez completado el bucle anterior, en la malla puede haber tetraedros transitorios que han sido marcados para refinar por el indicador de error (*Tipo I*) o que han recibido alguna marca en alguna de sus aristas. En estos casos, se va a realizar un estudio sobre los padres de los tetraedros transitorios implicados, teniendo en cuenta las tres posibilidades reseñadas anteriormente:

1. En caso de que el tetraedro transitorio esté señalado por el indicador de error, el tetraedro padre se marcará como refinable *Tipo I*.

2. En caso de que el tetraedro transitorio haya recibido una marca en una de las aristas propias (no compartida), se realizará el proceso anterior con el tetraedro padre (marcarlo como *Tipo I*).

3. Si la marca recibida es en una arista compartida con su tetraedro padre no se realizará la operación anterior. Simplemente el tetraedro padre pasará a la lista de tetraedros a estudiar.

Se empleará la notación e→$Padre$ para referenciar el elemento padre de e.

Algoritmo 3.4 Esquema del proceso del estudio de transitorios

```
1: función ESTUDIAR_TRANSITORIOS(T_k)
2:     hay_transitorios = false
3:     lista_padres_refinables = ∅
4:     lista_padres_a_estudiar = ∅
5:     para τ_i ∈ T_k
6:         si τ_i transitorio entonces
7:             si τ_i marcado por indicador de error entonces
8:                 lista_padres_refinables ←push τ_i→Padre
9:             si no
10:                si τ_i tiene aristas marcadas entonces
11:                    si aristas marcadas de τ_i compartidas con τ_i→Padre entonces
12:                        lista_padres_a_estudiar ←push τ_i→Padre
13:                    si no
14:                        lista_padres_refinables ←push τ_i→Padre
15:                    fin si
16:                fin si
17:            fin si
18:        fin si
19:    fin para
20:    si lista_padres_refinables ⋃ lista_padres_a_estudiar ≠ ∅ entonces
21:        hay_transitorios = true
22:        para τ_i ∈ lista_padres_refinables
23:            Marcar τ_i como refinable
24:            lista_padres_a_estudiar ←push τ_i
25:        fin para
26:        para τ_i ∈ lista_padres_a_estudiar
27:            DESHACER_TRANSITORIOS_TETRAEDRO(τ_i)
28:        fin para
29:        COMPACTAR_MALLA(T_k)
30:    fin si
31:    devolver hay_transitorios
32: fin función
```

Para realizar esta clasificación, se va a recorrer la lista de tetraedros y se preguntará si es transitorio y está marcado por el indicador (caso 1, línea 7 del algoritmo). En caso afirmativo se incluirá el padre del tetraedro en una lista de tetraedros a refinar. Si el tetraedro transitorio comparte las aristas marcadas con su padre (caso 3, línea 11), el padre del tetraedro se incluirá en una segunda lista de tetraedros, en este caso, a estudiar. Los transitorios marcados restantes (caso 2, línea 13) serán los que no compartan alguna arista con su tetraedro padre,

por lo que éste será incluido en la misma lista del caso 1, la de los tetraedros a refinar.

A los tetraedros incluidos en la lista de tetraedros a refinar se les marcará como afectados por el indicador de error (línea 23), o sea, de *Tipo I*, y serán incluidos en la lista de tetraedros a estudiar (línea 24). A todos los tetraedros incluidos en esta última lista (tanto los del caso 3 como los incluidos posteriormente) se les va a realizar un proceso de eliminación de su división interna, ya que son divisiones transitorias, y el hecho de marcar nuevos tetraedros como *Tipo I* provocará que alguno de los del caso 3 tengan que ser refinados de forma distinta a la original.

Este subproceso finaliza con una compactación de la malla para suprimir los elementos eliminados. Devolverá un valor **true** si se han encontrado transitorios marcados, lo que provocará un nuevo bucle en el proceso general de marcado (algoritmo 3.2).

3.3.3. Eliminación de transitorios

El proceso de eliminación de transitorios no va a realizar un borrado de las divisiones internas de los tetraedros. Esta tarea la realiza el proceso de compactación (sección 3.5).

A cada tetraedro que llegue al procedimiento (algoritmo 3.5) se le va a realizar una marca indicativa de que su división interna debe desaparecer. Teniendo en cuenta que estos tetraedros podrían recibir nuevas marcar en un proceso posterior de conformado, este marcado para eliminación generará un proceso recursivo a todos los tetraedros vecinos por caras, puesto que el hecho de recibir una nueva marca de conformidad o pasar a *Tipo I* implicará nuevos modos de división también en sus vecinos divididos en transitorios.

El proceso de eliminación de tetraedros va a generar una propagación por caras, haciendo que también sean marcadas para eliminar las caras divididas en caras transitorias. A su vez, el proceso de marcado para eliminación de caras llamará al de marcado de eliminación de tetraedros, lo que implicar un bucle recursivo entre dos procedimientos.

Puesto que en posteriores procesos de conformado siempre se añadirán nuevas marcas, el criterio de parada de la recursividad será encontrarse con una división permanente (tetraedros divididos en ocho subtetraedros o caras divididas en cuatro subtriángulos) o bien un tetraedro no dividido (no necesita

Algoritmo 3.5 Algoritmos de eliminación de transitorios

```
procedimiento DESHACER_TRANSITORIOS_TETRAEDRO(τ)
    si τ dividido en 8 ó no dividido entonces
        fin
    fin si
    si τ ya marcado para deshacer entonces
        fin
    fin si
    Marcar τ para deshacer división
    para χ_i ∈ C_τ
        DESHACER_TRANSITORIOS_CARA(χ_i)
    fin para
fin procedimiento

procedimiento DESHACER_TRANSITORIOS_CARA(χ)
    si χ dividido en 4 entonces
        fin
    fin si
    si χ ya marcado para deshacer entonces
        fin
    fin si
    para τ_i ∈ Ω_χ
        si χ está dividido entonces
            DESHACER_TRANSITORIOS_TETRAEDRO(τ_i)
        si no
            si τ_i es transitorio entonces
                DESHACER_TRANSITORIOS_TETRAEDRO(τ_i→Padre)
            fin si
        fin si
    fin para
    si χ dividida entonces
        Marcar χ para deshacer división
    fin si
fin procedimiento
```

propagar).

3.4. Proceso de división

Tras finalizar el proceso de marcado, la malla resultante tendrá sus aristas marcadas y estará conforme según los criterios del algoritmo de refinamiento. El siguiente paso será dividir los elementos marcados: aristas, caras y tetraedros. Puesto que cada elemento está *instanciado* sólo una vez en la malla (aunque referenciado múltiples veces desde otros elementos) sólo habrá que realizar una división por cada elemento.

En el proceso de división se realizará la creación de los elementos inter-

nos necesarios para generar los elementos hijos. En el proceso de compactación (sección 3.5) los nuevos elementos creados pasarán a formar parte de la malla refinada. Hasta entonces sólo podrán ser referenciados desde el propio elemento padre.

3.4.1. Paralelización del proceso de división

En el proceso de división de un elemento sólo es necesario tener en cuenta qué elementos lo componen, además de los nuevos elementos internos que se generen:

- la división de las aristas depende de sus nodos y del nodo interno que se cree
- la de las caras depende de sus aristas y de las aristas interiores que se creen
- la de los tetraedros depende de sus caras y de las aristas y caras que se generen

Por lo tanto, al ser cada elemento independiente del resto del mismo tipo y tener sus dependencias restringidas a sus elementos constructores se podría realizar una división en paralelo de todos ellos. Como en esta dependencia están implicados los elementos hijos, se realizará una división, en primer lugar, de todas las aristas, después de las caras y finalmente de los tetraedros. La paralelización se realizará a nivel de cada grupo de elementos.

Los procesos paralelos de división se han implementado empleando *threads* (sección 2.3). Al ejecutarse todos los procesos dentro de una misma máquina no es necesario tener en cuenta particiones de la malla ni generación de espacios de memoria independiente. Lo que sí es imprescindible es el control de concurrencia y se realizará mediante *mutexs* y variables de condición (secciones 2.3.2 y 2.3.3). El esquema implementado se corresponde con el algoritmo del productor-consumidor (algoritmo 2.3).

3.4.2. Lanzamiento de procesos

En el inicio del programa (algoritmo 3.6) se lanzan una serie de procesos que se ejecutan en paralelo. Para cada tipo de elemento (arista, cara y tetraedro)

se van lanzar tantos procesos como CPU's disponga la máquina en la que se ejecutan. Estos serán procesos consumidores: estarán a la espera de que hayan datos disponibles. Mientras no haya, quedarán suspendidos sin ocupar tiempo de CPU.

Algoritmo 3.6 Lanzamiento de procesos paralelos de división

```
 1: procedimiento CREACIÓN_PROCESOS_DIVISIÓN
 2:     lista_aristas_a_dividir = new IBuffer<Arista>
 3:     lista_caras_a_dividir = new IBuffer<Cara>
 4:     lista_tetraedros_a_dividir = new IBuffer<Tetraedro>
 5:     lista_procesos = ∅
 6:     para 1 hasta Numero_CPU's
 7:         lista_procesos <-push- new PIThread<PROCESO_DIVISIÓN_ARISTAS>
 8:         lista_procesos <-push- new PIThread<PROCESO_DIVISIÓN_CARAS>
 9:         lista_procesos <-push- new PIThread<PROCESO_DIVISIÓN_TETRAEDROS>
10:     fin para
11: fin procedimiento
```

La clase *IBuffer* (sección 2.4.3.2) implementa una cola de elementos con control de acceso concurrente. El proceso de división inserta elementos en la cola *IBuffer* y los procesos paralelos extraen elementos. Estas acciones se consideran secciones críticas, por lo que se ha usado un mutex para controlar su ejecución. Una variable de condición asociada al *IBuffer* es empleada por los procesos paralelos para ponerse en espera. Cuando no hay datos, el intento de lectura en el *IBuffer* hace que el proceso quede en estado "esperando", mientras que una inserción de un elemento por parte del proceso principal hará que despierten los procesos y comiencen a extraer elementos para ser divididos. En el proceso inicial se crean tres *IBuffer*, uno para cada tipo de elemento a dividir.

La clase *PIThread* (sección 2.4.3.2) encapsula las llamadas a la librería *POSIX* (sección 2.3.1) para la creación de procesos paralelos. Cada vez que se crea un proceso se añade a una lista de procesos en ejecución, de manera que el módulo principal pueda tener referencias y ordenar su parada al finalizar el programa.

3.4.3. Módulo principal de división

El proceso principal de división (algoritmo 3.7) se comporta como un proceso productor: pondrá a disposición de los procesos paralelos elementos a dividir. Cuando haya elementos disponibles, los procesos de división serán despertados

y podrán tomar uno de los elementos y proceder a su partición. Una vez que todos los elementos han sido divididos, el proceso principal puede continuar.

Inicialmente, se implementó la detección por parte del proceso principal de que todos los elementos han sido divididos poniéndolo en espera hasta que el *IBuffer* asociado al tipo de elemento estuviera vacío. Esto provocó errores de concurrencia: el módulo principal pasaba a una nueva fase de divisiones cuando el proceso paralelo aún no había terminado completamente la división. El proceso principal se encontraba con elementos a medio dividir. Este problema se solucionó mediante la incorporación de unos indicadores globales: cuando una determinada variable alcance un valor esperado se da el proceso por concluido.

El proceso principal inicializa un contador a cero y la variable de control al número de elementos que deben ser divididos. El proceso paralelo incrementará el contador cada ver que haga una división. Cuando el contador y la variable de control tengan el mismo valor, el proceso de división ha concluido su trabajo.

Algoritmo 3.7 Proceso principal de división

```
 1: procedimiento DIVISIÓN(Malla T_k)
 2:     control_aristas = 0
 3:     contador_aristas = 0
 4:     para α_i ∈ aristas de T_k
 5:         control_aristas = control_aristas + 1
 6:         lista_aristas_a_dividir <-push- α_i
 7:     fin para
 8:     esperar a control_aristas = contador_aristas
 9:     control_caras = 0
10:     contador_caras = 0
11:     para χ_i ∈ caras de T_k
12:         control_caras = control_caras + 1
13:         lista_caras_a_dividir <-push- χ_i
14:     fin para
15:     esperar a control_caras = contador_caras
16:     control_tetraedros = 0
17:     contador_tetraedros = 0
18:     para τ_i ∈ tetraedros de T_k
19:         control_tetraedros = control_tetraedros + 1
20:         lista_tetraedros_a_dividir <-push- τ_i
21:     fin para
22:     esperar a control_tetraedros = contador_tetraedros
23: fin procedimiento
```

3.4.4. Procesos paralelos

Los procesos paralelos (algoritmo 3.8) se comportan como consumidores. Los tres tipos de procesos tienen un esquema similar: mientras haya elementos en el *IBuffer* asociado, lo extraen, lanzan su módulo particular de división, incrementan el contador de elementos divididos y vuelven al principio.

Dentro del *IBuffer* existe un indicador de actividad. Cuando el programa concluye pondrá los *IBuffer* como inactivos, se despertarán los procesos paralelos y al comprobar que no hay actividad terminarán su ejecución.

Algoritmo 3.8 Esquema de un proceso paralelo de división

```
1:  procedimiento PROCESO_DIVISIÓN_Elemento
2:      mientras lista_elemento_a_dividir activo
3:          e ←pop lista_elemento_a_dividir
4:          si lista_elemento_a_dividir no activo entonces
5:              salir del bucle mientras
6:          fin si
7:          si e = ∅ entonces
8:              continuar bucle mientras
9:          fin si
10:         DIVIDIR(e)
11:         contador_elementos = contador_elementos + 1
12:     fin mientras
13: fin procedimiento
```

En la línea 3 se produce el bloqueo del proceso cuando al intentar leer no hay elementos. El hecho de que se añada un elemento o se marque el fin del *IBuffer* por el programa principal hará que se despierte el proceso paralelo. En ocasiones el sistema operativo provoca despertares aleatorios en los procesos, y esto se puede comprobar cuando el elemento devuelto por el *IBuffer* está vacío (línea 7).

3.4.5. Finalización de procesos

Finalmente, cuando el programa principal debe finalizar, desactivará los *IBuffer* despertando a los procesos paralelos. Esperará que estos concluyan (invocando un método *Join*) y podrá terminar su ejecución (algoritmo 3.9).

Algoritmo 3.9 Finalización de los procesos paralelos de división

```
1: procedimiento FINALIZACIÓN_PROCESOS_DIVISIÓN
2:     desactivar lista_aristas_a_dividir
3:     desactivar lista_caras_a_dividir
4:     desactivar lista_tetraedros_a_dividir
5:     mientras lista_procesos ≠ ∅
6:         p ←pop lista_procesos
7:         JOIN(p)
8:     fin mientras
9: fin procedimiento
```

3.4.6. Etiquetado de elementos

Los nodos son los elementos más sencillos de la malla: representan un punto en el espacio mediante sus coordenadas, y no se necesita más de ellos. A partir de los nodos podemos construir las aristas, desde las aristas, las caras y finalmente, los tetraedros.

Cada elemento, inicialmente, conoce cuales son sus elementos constructores. En el proceso de división se va a realizar un etiquetado de aquellos elementos que se van a generar nuevos. Estas etiquetas se mantendrán de forma local, es decir, internas a cada elemento. De esta forma, las referencias de una arista no tienen que ser iguales vista desde una cara que desde otra ya que cada una mantendrá su propio juego de marcas.

En general, los procesos de división comenzarán localizando qué elementos figuran como marcados, puesto que estos definen la forma de dividirlos. A partir de aquí, en cada proceso, se crearán las etiquetas necesarias para referenciar todos los subelementos.

3.4.6.1. Notación

Se va a emplear la siguiente notación para indicar elementos:

- Una letra mayúscula para los nodos propios del elemento
- Una letra minúscula para los nodos introducidos en las aristas
- El formato $\overline{AB}$ para indicar la arista formada por los nodos A y B.
- El formato $\widehat{ABC}$ para indicar la cara formada por los nodos A, B y C, que tendrá las aristas $\overline{AB}$, $\overline{AC}$ y $\overline{BC}$

Para especificar referencias locales se va a emplear notación tipo C. Así, dada una arista α, una cara χ y un tetraedro τ, tendremos:

- $\alpha \rightarrow A$ indica el nodo A de la arista α
- $\chi \rightarrow \overline{AB}$ señala la arista $\overline{AB}$ de la cara χ
- $\tau \rightarrow \widehat{ABC}$ señala la cara del tetraedro τ formada por los nodos A, B y C
- $\tau \rightarrow \boxed{\text{ABCD}}$ señalará un subtetraedro dentro de τ formado por los nodos A, B, C y D

Puesto que las referencias son locales, en cada elemento se podrá alterar el punto se vista de la referencia origen. De esta forma:

- $\left\langle \chi \rightarrow \overline{AB} \right\rangle_B$ devolverá la arista $\overline{AB}$ de la cara χ, pero tomando como referencia el nodo B
- $\left\langle \tau \rightarrow \widehat{ABC} \right\rangle_{(B,C,A)}$ devolverá la cara $\widehat{ABC}$ del tetraedro τ, pero cambiando las referencias y tomando como primer nodo B y como segundo C

El hecho de cambiar una referencia local implicará que se modifiquen adecuadamente todos los marcadores internos que se agreguen en un proceso de división. En cada uno de esos procesos se mostrará qué referencias se van a definir.

3.4.7. División de aristas

Dada una arista $\overline{MN}$ (M y N nodos) que ha sido marcada durante el proceso de refinamiento, el algoritmo 3.10 presenta los pasos para obtener la división mostrada en la figura 3.5(a).

Algoritmo 3.10 División de una arista

```
1: procedimiento DIVIDIR_ARISTA(α)
2:     α→i = nueva nodo((M+N)/2)
3:     α→Mi = nueva arista(M,i)
4:     α→Ni = nueva arista(N,i)
5: fin procedimiento
```

En caso de que se realice una referencia del tipo $\langle \alpha \rangle_N$, la figura 3.5(b) muestra como quedaría el etiquetado correspondiente.

Figura 3.5: División de una arista

3.4.8. División de caras

Dada una cara χ, etiquetada inicialmente como muestra la figura 3.6, se va a realizar la división de la misma en función de qué aristas tiene marcadas. Partiendo de esas aristas, y realizando las reorientaciones necesarias en cada caso, se obtendrán las diferentes etiquetas según el proceso de división empleado.

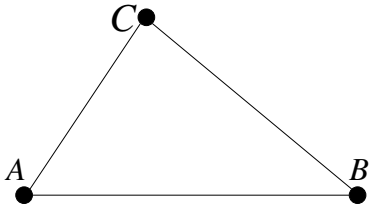


Figura 3.6: Etiquetas iniciales de una cara

3.4.8.1. Cara con una arista marcada

Este es el modo más simple de división. Se parte la cara en dos, tal como muestra la figura 3.7. El algoritmo 3.11 detalla el proceso.

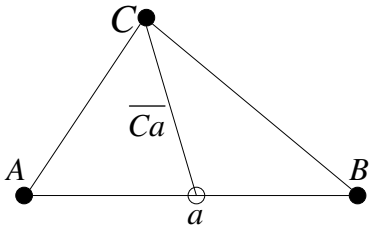


Figura 3.7: División de una cara en dos subtriángulos

Para indicar la lista de aristas no marcadas de una cara se volverá a emplear el operador de conjunto A'_χ, mientras que A^*_χ indicará las aristas marcadas.

Algoritmo 3.11 División de una cara en dos subtriángulos

1: **procedimiento** DIVIDIR_CARA_EN_2(χ)
2: $\chi{\rightarrow}\overline{AB} = A^*_\chi$ ▷ arista marcada
3: $\chi{\rightarrow}A = \chi{\rightarrow}\overline{AB}{\rightarrow}M$
4: $\chi{\rightarrow}B = \chi{\rightarrow}\overline{AB}{\rightarrow}N$
5: $\chi{\rightarrow}C = N_\chi$ - $N_{\chi{\rightarrow}\overline{AB}}$ ▷ Nodo opuesto a $\chi{\rightarrow}\overline{AB}$
6: $\rho = A'_\chi$ ▷ aristas no marcadas
7: **si** $N_{\rho_0} = \{\chi \rightarrow A, \chi \rightarrow C\}$ **entonces**
8: $\chi{\rightarrow}\overline{AC} = \langle\rho_0\rangle_{\chi{\rightarrow}A}$
9: $\chi{\rightarrow}\overline{BC} = \langle\rho_1\rangle_{\chi{\rightarrow}B}$
10: **si no**
11: $\chi{\rightarrow}\overline{AC} = \langle\rho_1\rangle_{\chi{\rightarrow}A}$
12: $\chi{\rightarrow}\overline{BC} = \langle\rho_0\rangle_{\chi{\rightarrow}B}$
13: **fin si**
14: $\chi{\rightarrow}\overline{Ca}$ = **nueva arista**($\chi{\rightarrow}C$, $\chi{\rightarrow}\overline{AB}{\rightarrow}i$)
15: $\chi{\rightarrow}\widehat{ACa}$ = **nueva cara**($\chi{\rightarrow}\overline{AB}{\rightarrow}\overline{Mi}$,$\chi{\rightarrow}\overline{AC}$,$\chi{\rightarrow}\overline{Ca}$)
16: $\chi{\rightarrow}\widehat{BCa}$ = **nueva cara**($\chi{\rightarrow}\overline{AB}{\rightarrow}\overline{Ni}$,$\chi{\rightarrow}\overline{BC}$,$\chi{\rightarrow}\overline{Ca}$)
17: **fin procedimiento**

3.4.8.2. Cara con dos aristas marcadas

En este caso se va a dividir la cara en tres subtriángulos, uniendo los dos nuevos nodos y dividiendo por la arista más larga, tal como muestra la figura 3.8. El algoritmo 3.12 muestra el proceso.

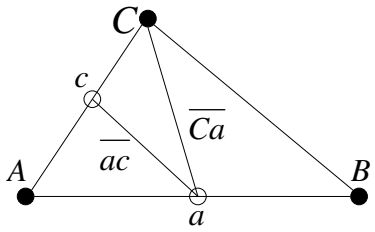


Figura 3.8: División de una cara en tres subtriángulos

3.4.8.3. Cara con tres aristas marcadas

Uniendo todos los nodos introducidos se generará un triángulo interior (figura 3.9) que dividirá en cuatro subtriángulos la cara marcada, tal como muestra el algoritmo 3.13.

Algoritmo 3.12 División de una cara en tres subtriángulos

1: **procedimiento** DIVIDIR_CARA_EN_3(χ)
2: $\quad\chi{\rightarrow}\overline{BC} = A'_\chi$ ▷ arista no marcada
3: $\quad\chi{\rightarrow}A = N_\chi$ - $N_{\chi\rightarrow\overline{BC}}$
4: $\quad\rho = A^*_\chi$ ▷ aristas marcadas
5: $\quad$**si** $|\rho_0| > |\rho_1|$ **entonces** ▷ comparar longitudes
6: $\quad\quad\chi{\rightarrow}\overline{AB} = \langle\rho_0\rangle_{\chi\rightarrow A}$
7: $\quad\quad\chi{\rightarrow}\overline{AC} = \langle\rho_1\rangle_{\chi\rightarrow A}$
8: $\quad$**si no**
9: $\quad\quad\chi{\rightarrow}\overline{AB} = \langle\rho_1\rangle_{\chi\rightarrow A}$
10: $\quad\quad\chi{\rightarrow}\overline{AC} = \langle\rho_0\rangle_{\chi\rightarrow A}$
11: $\quad$**fin si**
12: $\quad\chi{\rightarrow}\overline{ac}$ = **nueva arista**($\chi{\rightarrow}\overline{AB}{\rightarrow}i$, $\chi{\rightarrow}\overline{AC}{\rightarrow}i$)
13: $\quad\chi{\rightarrow}\overline{Ca}$ = **nueva arista**($\chi{\rightarrow}\overline{AB}{\rightarrow}i$, $\chi{\rightarrow}\overline{AC}{\rightarrow}N$)
14: $\quad\chi{\rightarrow}\widehat{Aac}$ = **nueva cara**($\chi{\rightarrow}\overline{AB}{\rightarrow}\overline{Mi}$, $\chi{\rightarrow}\overline{AC}{\rightarrow}\overline{Mi}$,$\chi{\rightarrow}\overline{ac}$)
15: $\quad\chi{\rightarrow}\widehat{Cac}$ = **nueva cara**($\chi{\rightarrow}\overline{Ca}$, $\chi{\rightarrow}\overline{AC}{\rightarrow}\overline{Ni}$,$\chi{\rightarrow}\overline{ac}$)
16: $\quad\chi{\rightarrow}\widehat{BCa}$ = **nueva cara**($\chi{\rightarrow}\overline{BC}$, $\chi{\rightarrow}\overline{AB}{\rightarrow}\overline{Ni}$,$\chi{\rightarrow}\overline{Ca}$)
17: **fin procedimiento**

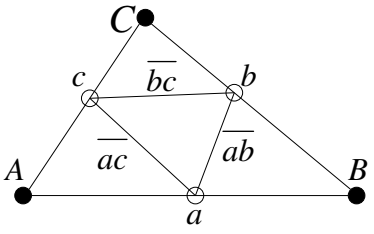


Figura 3.9: División de una cara en cuatro subtriángulos

Algoritmo 3.13 División de una cara en cuatro subtriángulos

procedimiento DIVIDIR_CARA_EN_4(χ)
  $\rho = A_\chi$ ▷ todas las aristas
  $\chi \rightarrow \overline{AB} = \rho_0$ ▷ tomar una cualquiera
  si $\chi \rightarrow \overline{AB} \rightarrow M \in N_{\rho_1}$ **entonces** ▷ buscar vecino
    $\chi \rightarrow \overline{AC} = \langle \rho_1 \rangle_{\chi \rightarrow \overline{AB} \rightarrow M}$
    $\chi \rightarrow \overline{BC} = \langle \rho_2 \rangle_{\chi \rightarrow \overline{AB} \rightarrow N}$
  si no
    $\chi \rightarrow \overline{AC} = \langle \rho_2 \rangle_{\chi \rightarrow \overline{AB} \rightarrow M}$
    $\chi \rightarrow \overline{BC} = \langle \rho_1 \rangle_{\chi \rightarrow \overline{AB} \rightarrow N}$
  fin si
  $\chi \rightarrow \overline{ab}$ = **nueva arista**($\chi \rightarrow \overline{AB} \rightarrow i$, $\chi \rightarrow \overline{BC} \rightarrow i$)
  $\chi \rightarrow \overline{bc}$ = **nueva arista**($\chi \rightarrow \overline{BC} \rightarrow i$, $\chi \rightarrow \overline{AC} \rightarrow i$)
  $\chi \rightarrow \overline{ac}$ = **nueva arista**($\chi \rightarrow \overline{AB} \rightarrow i$, $\chi \rightarrow \overline{AC} \rightarrow i$)
  $\chi \rightarrow \widehat{Aac}$ = **nueva cara**($\chi \rightarrow \overline{AB} \rightarrow \overline{Mi}$, $\chi \rightarrow \overline{AC} \rightarrow \overline{Mi}$,$\chi \rightarrow \overline{ac}$)
  $\chi \rightarrow \widehat{Bab}$ = **nueva cara**($\chi \rightarrow \overline{AB} \rightarrow \overline{Ni}$, $\chi \rightarrow \overline{BC} \rightarrow \overline{Mi}$,$\chi \rightarrow \overline{ab}$)
  $\chi \rightarrow \widehat{Cbc}$ = **nueva cara**($\chi \rightarrow \overline{BC} \rightarrow \overline{Ni}$, $\chi \rightarrow \overline{AC} \rightarrow \overline{Ni}$,$\chi \rightarrow \overline{bc}$)
  $\chi \rightarrow \widehat{abc}$ = **nueva cara**($\chi \rightarrow \overline{ab}$,$\chi \rightarrow \overline{bc}$,$\chi \rightarrow \overline{ac}$)
fin procedimiento

3.4.9. División de tetraedros

Dado un tetraedro τ, etiquetado inicialmente como muestra la figura 3.10, se procederá a su división en función del número y la posición de las arista marcadas.

De forma similar a los casos de división anteriormente expuestos se van a emplear los operadores C^*_τ para recuperar las caras marcadas de un tetraedro τ y el operador C'_τ para las caras no marcadas.

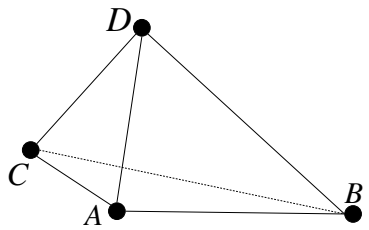


Figura 3.10: Etiquetas iniciales de un tetraedro

3.4.9.1. Tetraedro con una arista marcada

Se procederá a la bisección del tetraedro, tal como muestra la figura 3.11. El algoritmo 3.14 muestra el proceso.

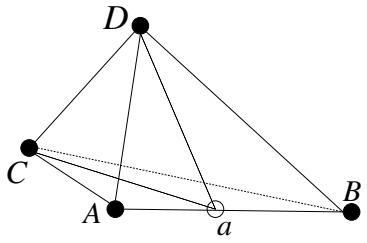


Figura 3.11: División de un tetraedro en dos subtetraedros (*Tipo IV*)

Algoritmo 3.14 División de un tetraedro en dos subtetraedros (*Tipo IV*)

procedimiento DIVIDIR_TETRAEDRO_EN_2(τ)
  $\omega = C^*_\tau$ ▷ obtener caras marcadas
  $\tau{\rightarrow}\widehat{ABC} = \omega_0$ ▷ tomar una cualquiera
  $\tau{\rightarrow}A = \tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}M$
  $\tau{\rightarrow}B = \tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}N$
  $\tau{\rightarrow}C = \tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AC}{\rightarrow}N$
  $\tau{\rightarrow}D = N_\tau$ - $N_{\tau{\rightarrow}\widehat{ABC}}$
  $\tau{\rightarrow}\widehat{ABD} = \langle\omega_1\rangle_{(\tau{\rightarrow}A,\tau{\rightarrow}B,\tau{\rightarrow}D)}$
  $\tau{\rightarrow}\overline{CD} = A_\tau$ - $A_{\tau{\rightarrow}\widehat{ABC}}$ - $A_{\tau{\rightarrow}\widehat{ABD}}$
  $\lambda = C'_\tau$ ▷ caras no marcadas
  si $N_{\lambda_0} = \{\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D\}$ **entonces**
    $\tau{\rightarrow}\widehat{ACD} = \lambda_0$
    $\tau{\rightarrow}\widehat{BCD} = \lambda_1$
  si no
    $\tau{\rightarrow}\widehat{ACD} = \lambda_1$
    $\tau{\rightarrow}\widehat{BCD} = \lambda_0$
  fin si
  $\tau{\rightarrow}\widehat{CDa}$ = **nueva cara**($\tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{Ca}$, $\tau{\rightarrow}\widehat{ABD}{\rightarrow}\overline{Ca}$,$\tau{\rightarrow}\overline{CD}$)
  $\tau{\rightarrow}\boxed{\text{ACDa}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{ACD}$, $\tau{\rightarrow}\widehat{ABD}{\rightarrow}\widehat{ACa}$, $\tau{\rightarrow}\widehat{ABC}{\rightarrow}\widehat{ACa}$,$\tau{\rightarrow}\widehat{CDa}$)
  $\tau{\rightarrow}\boxed{\text{BCDa}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{BCD}$, $\tau{\rightarrow}\widehat{ABD}{\rightarrow}\widehat{BCa}$, $\tau{\rightarrow}\widehat{ABC}{\rightarrow}\widehat{BCa}$,$\tau{\rightarrow}\widehat{CDa}$)
fin procedimiento

3.4.9.2. Tetraedro con dos aristas marcadas en la misma cara

Habrá que generar dos aristas interiores que dividan el tetraedro en tres subtetraedros (figura 3.12 y algoritmo 3.15).

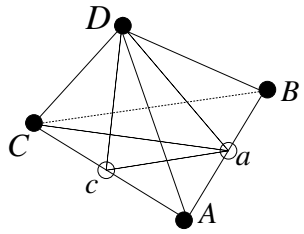


Figura 3.12: División de un tetraedro en tres subtetraedros (*Tipo IIIb*)

Algoritmo 3.15 División de un tetraedro en tres subtetraedros (*Tipo IIIb*)

1: **procedimiento** DIVIDIR_TETRAEDRO_EN_3(τ)
2: $\tau \rightarrow \widehat{BCD} = C'_\tau$ ▷ cara no marcada
3: **para** $\chi_i \in C^*_\tau$ ▷ buscar en caras marcadas
4: **si** χ_i tiene dos marcas **entonces**
5: $\tau \rightarrow \widehat{ABC} = \chi_i$
6: **fin si**
7: **fin para**
8: $\tau \rightarrow A = \tau \rightarrow \widehat{ABC} \rightarrow \overline{AB} \rightarrow M$
9: $\tau \rightarrow B = \tau \rightarrow \widehat{ABC} \rightarrow \overline{AB} \rightarrow N$
10: $\tau \rightarrow C = \tau \rightarrow \widehat{ABC} \rightarrow \overline{AC} \rightarrow N$
11: $\tau \rightarrow D = N_\tau$ - $N_{\tau \rightarrow \widehat{ABC}}$
12: $\omega = C^*_\tau$ - $\tau \rightarrow \widehat{ABC}$ ▷ obtener caras con 1 marca
13: **si** $N_{\omega_0} = \{\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D\}$ **entonces**
14: $\tau \rightarrow \widehat{ABD} = \langle \omega_0 \rangle_{(\tau \rightarrow A, \tau \rightarrow B, \tau \rightarrow D)}$
15: $\tau \rightarrow \widehat{ACD} = \langle \omega_1 \rangle_{(\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D)}$
16: **si no**
17: $\tau \rightarrow \widehat{ABD} = \langle \omega_1 \rangle_{(\tau \rightarrow A, \tau \rightarrow B, \tau \rightarrow D)}$
18: $\tau \rightarrow \widehat{ACD} = \langle \omega_0 \rangle_{(\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D)}$
19: **fin si**
20: $\tau \rightarrow \widehat{Dac}$ = **nueva cara**($\tau \rightarrow \widehat{ABC} \rightarrow \overline{ac}$, $\tau \rightarrow \widehat{ABD} \rightarrow \overline{Ca}$, $\tau \rightarrow \widehat{ACD} \rightarrow \overline{Ca}$)
21: $\tau \rightarrow \widehat{CDa}$ = **nueva cara**($\tau \rightarrow \widehat{ABC} \rightarrow \overline{Ca}$, $\tau \rightarrow \widehat{ABD} \rightarrow \overline{Ca}$, $\tau \rightarrow \widehat{ACD} \rightarrow \overline{BC}$)
22: $\tau \rightarrow \boxed{\text{ADac}}$ = **nuevo tet**($\tau \rightarrow \widehat{ABC} \rightarrow \widehat{Aac}$, $\tau \rightarrow \widehat{ABD} \rightarrow \widehat{ACa}$, $\tau \rightarrow \widehat{ACD} \rightarrow \widehat{ACa}$,$\tau \rightarrow \widehat{Dac}$)
23: $\tau \rightarrow \boxed{\text{CDca}}$ = **nuevo tet**($\tau \rightarrow \widehat{ABC} \rightarrow \widehat{Cac}$, $\tau \rightarrow \widehat{ACD} \rightarrow \widehat{BCa}$,$\tau \rightarrow \widehat{CDa}$,$\tau \rightarrow \widehat{Dac}$)
24: $\tau \rightarrow \boxed{\text{BCDa}}$ = **nuevo tet**($\tau \rightarrow \widehat{ABC} \rightarrow \widehat{BCa}$, $\tau \rightarrow \widehat{ABD} \rightarrow \widehat{BCa}$,$\tau \rightarrow \widehat{BCD}$,$\tau \rightarrow \widehat{CDa}$)
25: **fin procedimiento**

3.4.9.3. Tetraedro con dos aristas marcadas en distinta cara

Habrá que generar una arista interior, que dividirá en cuatro subtetraedros el elemento principal (figura 3.13 y algoritmo 3.16).

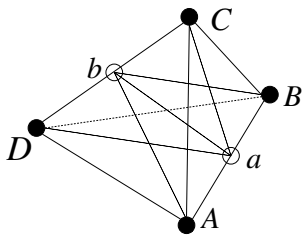


Figura 3.13: División de un tetraedro en cuatro subtetraedros (*Tipo IIIa*)

Algoritmo 3.16 División de un tetraedro en cuatro subtetraedros (*Tipo IIIa*)

procedimiento DIVIDIR_TETRAEDRO_EN_4A(τ)
  $\omega = C_\tau$ ▷ Caras marcadas (todas)
  $\tau{\rightarrow}\widehat{ABC} = \omega_0$ ▷ Tomar una cara cualquiera
  $\tau{\rightarrow}A = \tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}M$
  $\tau{\rightarrow}B = \tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}N$
  $\tau{\rightarrow}C = \tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AC}{\rightarrow}N$
  $\tau{\rightarrow}D = N_\tau$ - $N_{\tau\rightarrow\widehat{ABC}}$
  para $\omega_i \in C_\tau$ ▷ localizar resto de caras
    si $N_{\omega_i} = \{\tau \rightarrow A, \tau \rightarrow B, \tau \rightarrow D\}$ **entonces**
      $\tau{\rightarrow}\widehat{ABD} = \langle\omega_i\rangle_{\tau\rightarrow A,\tau\rightarrow B,\tau\rightarrow D}$
    si no si $N_{\omega_i} = \{\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D\}$ **entonces**
      $\tau{\rightarrow}\widehat{CDA} = \langle\omega_i\rangle_{\tau\rightarrow C,\tau\rightarrow D,\tau\rightarrow A}$
    si no si $N_{\omega_i} = \{\tau \rightarrow B, \tau \rightarrow C, \tau \rightarrow D\}$ **entonces**
      $\tau{\rightarrow}\widehat{CDB} = \langle\omega_i\rangle_{\tau\rightarrow C,\tau\rightarrow D,\tau\rightarrow B}$
    fin si
  fin para
  $\tau{\rightarrow}\overline{ab}$ = **nueva arista**($\tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}i$,$\tau{\rightarrow}\widehat{CDB}{\rightarrow}\overline{AB}{\rightarrow}i$)
  $\tau{\rightarrow}\widehat{Aab}$ = **nueva cara**($\tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}\overline{Mi}$, $\tau{\rightarrow}\widehat{CDA}{\rightarrow}\overline{Ca}$,$\tau{\rightarrow}\overline{ab}$)
  $\tau{\rightarrow}\widehat{Bab}$ = **nueva cara**($\tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{AB}{\rightarrow}\overline{Ni}$, $\tau{\rightarrow}\widehat{CDB}{\rightarrow}\overline{Ca}$,$\tau{\rightarrow}\overline{ab}$)
  $\tau{\rightarrow}\widehat{Cab}$ = **nueva cara**($\tau{\rightarrow}\widehat{CDB}{\rightarrow}\overline{AB}{\rightarrow}\overline{Mi}$, $\tau{\rightarrow}\widehat{ABC}{\rightarrow}\overline{Ca}$,$\tau{\rightarrow}\overline{ab}$)
  $\tau{\rightarrow}\widehat{Dab}$ = **nueva cara**($\tau{\rightarrow}\widehat{CDB}{\rightarrow}\overline{AB}{\rightarrow}\overline{Ni}$, $\tau{\rightarrow}\widehat{ABD}{\rightarrow}\overline{Ca}$,$\tau{\rightarrow}\overline{ab}$)
  $\tau{\rightarrow}\boxed{\text{ADab}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{ABD}{\rightarrow}\widehat{ACa}$, $\tau{\rightarrow}\widehat{CDA}{\rightarrow}\widehat{BCa}$,$\tau{\rightarrow}\widehat{Aab}$,$\tau{\rightarrow}\widehat{Dab}$)
  $\tau{\rightarrow}\boxed{\text{BDab}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{ABD}{\rightarrow}\widehat{BCa}$, $\tau{\rightarrow}\widehat{CDB}{\rightarrow}\widehat{BCa}$,$\tau{\rightarrow}\widehat{Bab}$,$\tau{\rightarrow}\widehat{Dab}$)
  $\tau{\rightarrow}\boxed{\text{ACab}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{ABC}{\rightarrow}\widehat{ACa}$, $\tau{\rightarrow}\widehat{CDA}{\rightarrow}\widehat{ACa}$,$\tau{\rightarrow}\widehat{Aab}$,$\tau{\rightarrow}\widehat{Cab}$)
  $\tau{\rightarrow}\boxed{\text{BCab}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{ABC}{\rightarrow}\widehat{BCa}$, $\tau{\rightarrow}\widehat{CDB}{\rightarrow}\widehat{ACa}$,$\tau{\rightarrow}\widehat{Bab}$,$\tau{\rightarrow}\widehat{Cab}$)
fin procedimiento

3.4.9.4. Tetraedro con tres aristas marcadas en la misma cara

Habrá que generar tres aristas interiores para dividir el tetraedro en cuatro subtetraedros (figura 3.14 y algoritmo 3.17).

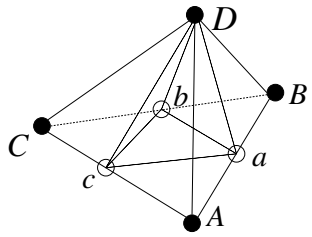


Figura 3.14: División de un tetraedro en cuatro subtetraedros (*Tipo II*)

Algoritmo 3.17 División de un tetraedro en cuatro subtetraedros (*Tipo II*)

procedimiento DIVIDIR_TETRAEDRO_EN_4B(τ)
  para $\omega_i \in C_\tau$
    si ω_i tiene 3 marcas **entonces** ▷ localizar cara con 3 marcas
      $\tau \rightarrow \widehat{ABC} = \omega_i$
    fin si
  fin para
  $\tau \rightarrow A = \tau \rightarrow \widehat{ABC} \rightarrow \overline{AB} \rightarrow M$
  $\tau \rightarrow B = \tau \rightarrow \widehat{ABC} \rightarrow \overline{AB} \rightarrow N$
  $\tau \rightarrow C = \tau \rightarrow \widehat{ABC} \rightarrow \overline{AC} \rightarrow N$
  $\tau \rightarrow D = N_\tau$ - $N_{\tau \rightarrow \widehat{ABC}}$
  para $\omega_i \in \left[C_\tau - \tau \rightarrow \widehat{ABC}\right]$ ▷ caras con una marca
    si $\tau \rightarrow \widehat{ABC} \rightarrow \overline{AB} \in A_{\omega_i}$ **entonces**
      $\tau \rightarrow \widehat{ABD} = \langle \omega_i \rangle_{\tau \rightarrow A, \tau \rightarrow B, \tau \rightarrow D}$
    si no si $\tau \rightarrow \widehat{ABC} \rightarrow \overline{BC} \in A_{\omega_i}$ **entonces**
      $\tau \rightarrow \widehat{BCD} = \langle \omega_i \rangle_{\tau \rightarrow B, \tau \rightarrow C, \tau \rightarrow D}$
    si no
      $\tau \rightarrow \widehat{ACD} = \langle \omega_i \rangle_{\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D}$
    fin si
  fin para
  $\tau \rightarrow \widehat{Dab}$ = **nueva cara**($\tau \rightarrow \widehat{ABC} \rightarrow \overline{ab}$, $\tau \rightarrow \widehat{ABD} \rightarrow \overline{Ca}$, $\tau \rightarrow \widehat{BCD} \rightarrow \overline{Ca}$)
  $\tau \rightarrow \widehat{Dac}$ = **nueva cara**($\tau \rightarrow \widehat{ABC} \rightarrow \overline{ac}$, $\tau \rightarrow \widehat{ABD} \rightarrow \overline{Ca}$, $\tau \rightarrow \widehat{ACD} \rightarrow \overline{Ca}$)
  $\tau \rightarrow \widehat{Dbc}$ = **nueva cara**($\tau \rightarrow \widehat{ABC} \rightarrow \overline{bc}$, $\tau \rightarrow \widehat{BCD} \rightarrow \overline{Ca}$, $\tau \rightarrow \widehat{ACD} \rightarrow \overline{Ca}$)
  $\tau \rightarrow \boxed{\text{ADac}}$ = **nuevo tet**($\tau \rightarrow \widehat{ABD} \rightarrow \widehat{ACa}$, $\tau \rightarrow \widehat{ACD} \rightarrow \widehat{ACa}$, $\tau \rightarrow \widehat{ABC} \rightarrow \widehat{Aac}$,$\tau \rightarrow \widehat{Dac}$)
  $\tau \rightarrow \boxed{\text{BDab}}$ = **nuevo tet**($\tau \rightarrow \widehat{ABD} \rightarrow \widehat{BCa}$, $\tau \rightarrow \widehat{BCD} \rightarrow \widehat{ACa}$, $\tau \rightarrow \widehat{ABC} \rightarrow \widehat{Bab}$,$\tau \rightarrow \widehat{Dab}$)
  $\tau \rightarrow \boxed{\text{CDbc}}$ = **nuevo tet**($\tau \rightarrow \widehat{ACD} \rightarrow \widehat{BCa}$, $\tau \rightarrow \widehat{BCD} \rightarrow \widehat{BCa}$, $\tau \rightarrow \widehat{ABC} \rightarrow \widehat{Cbc}$,$\tau \rightarrow \widehat{Dbc}$)
  $\tau \rightarrow \boxed{\text{Dabc}}$ = **nuevo tet**($\tau \rightarrow \widehat{ABC} \rightarrow \widehat{abc}$,$\tau \rightarrow \widehat{Dac}$,$\tau \rightarrow \widehat{Dab}$,$\tau \rightarrow \widehat{Dbc}$)
fin procedimiento

3.4.9.5. Tetraedro con las seis aristas marcadas

En primer lugar se van a generar cuatro caras internas, y se dividirá el tetraedro en cuatro subtetraedros (los coincidentes con sus vértices), según la figura 3.15.

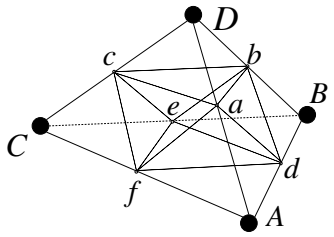


Figura 3.15: División de un tetraedro en ocuatro subtetraedros y un octaedro interior (*Tipo I*)

Para el octaedro interior (figura 3.16) que resulta de la división habrá tres posibilidades, en función de la distancia entre las tres parejas de nodos opuestos. Se realizará un reetiquetado en función de la distancia mínima, como puede verse en la figura 3.17. En función de los nodos elegidos, se generará una arista interior que dividirá el octaedro en otros cuatro subtetraedros.

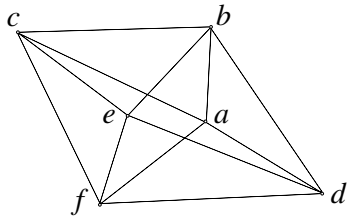
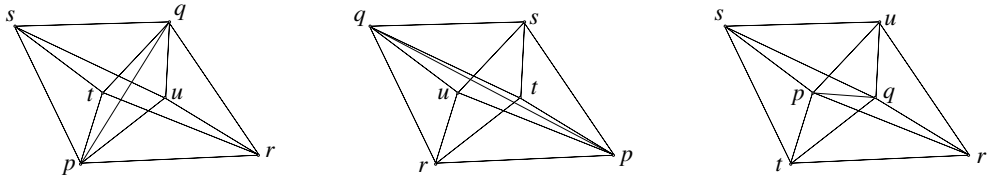


Figura 3.16: Octaedro resultante de la división *Tipo I*



(a) mín = $|\overline{bf}|$ (b) mín = $|\overline{cd}|$ (c) mín = $|\overline{ae}|$

Figura 3.17: Etiquetado del octaedro en función de la distancia mínima

Algoritmo 3.18 División de un tetraedro en ocho subtetraedros (*Tipo I*)

procedimiento DIVIDIR_TETRAEDRO_EN_8(τ)
  $\omega = N_\tau$
  $\tau{\rightarrow}A = \omega_0$ ▷ se toman los nodos en cualquier orden
  $\tau{\rightarrow}B = \omega_1$
  $\tau{\rightarrow}C = \omega_2$
  $\tau{\rightarrow}D = \omega_3$
  para $\chi_i \in C_\tau$
    si $N_{\chi_i} = \{\tau \rightarrow A, \tau \rightarrow B, \tau \rightarrow C\}$ **entonces**
      $\tau{\rightarrow}\widehat{ABC} = \langle\chi_i\rangle_{\tau\rightarrow A,\tau\rightarrow B,\tau\rightarrow C}$
    si no si $N_{\chi_i} = \{\tau \rightarrow A, \tau \rightarrow B, \tau \rightarrow D\}$ **entonces**
      $\tau{\rightarrow}\widehat{ABD} = \langle\chi_i\rangle_{\tau\rightarrow A,\tau\rightarrow B,\tau\rightarrow D}$
    si no si $N_{\chi_i} = \{\tau \rightarrow A, \tau \rightarrow C, \tau \rightarrow D\}$ **entonces**
      $\tau{\rightarrow}\widehat{ACD} = \langle\chi_i\rangle_{\tau\rightarrow A,\tau\rightarrow C,\tau\rightarrow D}$
    si no si $N_{\chi_i} = \{\tau \rightarrow B, \tau \rightarrow C, \tau \rightarrow D\}$ **entonces**
      $\tau{\rightarrow}\widehat{BCD} = \langle\chi_i\rangle_{\tau\rightarrow B,\tau\rightarrow C,\tau\rightarrow D}$
    fin si
  fin para
  ASIGNA_DIVIDIR_TETRAEDRO_EN_8(τ)
  $\tau{\rightarrow}\widehat{abc}$ = **nueva cara**($\tau{\rightarrow}\overline{ab}$,$\tau{\rightarrow}\overline{ac}$,$\tau{\rightarrow}\overline{bc}$)
  $\tau{\rightarrow}\widehat{adf}$ = **nueva cara**($\tau{\rightarrow}\overline{ad}$,$\tau{\rightarrow}\overline{af}$,$\tau{\rightarrow}\overline{df}$)
  $\tau{\rightarrow}\widehat{cef}$ = **nueva cara**($\tau{\rightarrow}\overline{ce}$,$\tau{\rightarrow}\overline{cf}$,$\tau{\rightarrow}\overline{ef}$)
  $\tau{\rightarrow}\widehat{bde}$ = **nueva cara**($\tau{\rightarrow}\overline{bd}$,$\tau{\rightarrow}\overline{be}$,$\tau{\rightarrow}\overline{de}$)
  $\tau{\rightarrow}\boxed{\text{Dabc}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{abc}$,$\tau{\rightarrow}\widehat{Dab}$,$\tau{\rightarrow}\widehat{Dac}$,$\tau{\rightarrow}\widehat{Dbc}$)
  $\tau{\rightarrow}\boxed{\text{Aadf}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{adf}$,$\tau{\rightarrow}\widehat{Aaf}$,$\tau{\rightarrow}\widehat{Adf}$,$\tau{\rightarrow}\widehat{Aad}$)
  $\tau{\rightarrow}\boxed{\text{Ccef}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{cef}$,$\tau{\rightarrow}\widehat{Ccf}$,$\tau{\rightarrow}\widehat{Cef}$,$\tau{\rightarrow}\widehat{Cce}$)
  $\tau{\rightarrow}\boxed{\text{Bbde}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{bde}$,$\tau{\rightarrow}\widehat{Bbe}$,$\tau{\rightarrow}\widehat{Bde}$,$\tau{\rightarrow}\widehat{Bbd}$)
  ORIENTA_OCTAEDRO(τ)
  $\tau{\rightarrow}\overline{pq}$ = **nueva arista**($\tau{\rightarrow}p$,$\tau{\rightarrow}q$)
  $\tau{\rightarrow}\widehat{pqr}$ = **nueva cara**($\tau{\rightarrow}\overline{pq}$,$\tau{\rightarrow}\overline{pr}$,$\tau{\rightarrow}\overline{qr}$)
  $\tau{\rightarrow}\widehat{pqs}$ = **nueva cara**($\tau{\rightarrow}\overline{pq}$,$\tau{\rightarrow}\overline{ps}$,$\tau{\rightarrow}\overline{qs}$)
  $\tau{\rightarrow}\widehat{pqu}$ = **nueva cara**($\tau{\rightarrow}\overline{pq}$,$\tau{\rightarrow}\overline{pu}$,$\tau{\rightarrow}\overline{qu}$)
  $\tau{\rightarrow}\widehat{pqt}$ = **nueva cara**($\tau{\rightarrow}\overline{pq}$,$\tau{\rightarrow}\overline{pr}$,$\tau{\rightarrow}\overline{qr}$)
  $\tau{\rightarrow}\boxed{\text{pqrt}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{pqr}$,$\tau{\rightarrow}\widehat{pqt}$,$\tau{\rightarrow}\widehat{prt}$,$\tau{\rightarrow}\widehat{qrt}$)
  $\tau{\rightarrow}\boxed{\text{pqst}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{pqs}$,$\tau{\rightarrow}\widehat{pqt}$,$\tau{\rightarrow}\widehat{pst}$,$\tau{\rightarrow}\widehat{qst}$)
  $\tau{\rightarrow}\boxed{\text{pqru}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{pqr}$,$\tau{\rightarrow}\widehat{pqu}$,$\tau{\rightarrow}\widehat{pru}$,$\tau{\rightarrow}\widehat{qru}$)
  $\tau{\rightarrow}\boxed{\text{pqsu}}$ = **nuevo tet**($\tau{\rightarrow}\widehat{pqs}$,$\tau{\rightarrow}\widehat{pqu}$,$\tau{\rightarrow}\widehat{psu}$,$\tau{\rightarrow}\widehat{qsu}$)
fin procedimiento

Algoritmo 3.19 Asignaciones para la división en ocho subtetraedros (*Tipo I*)

1: **procedimiento** ASIGNA_DIVIDIR_TETRAEDRO_EN_8(τ)

2:
$\tau\to f=\tau\to\widehat{ACD}\to\overline{AB}\to i$ | $\tau\to c=\tau\to\widehat{ACD}\to\overline{BC}\to i$
$\tau\to a=\tau\to\widehat{ACD}\to\overline{AC}\to i$ | $\tau\to d=\tau\to\widehat{ABC}\to\overline{AB}\to i$
$\tau\to e=\tau\to\widehat{ABC}\to\overline{BC}\to i$ | $\tau\to b=\tau\to\widehat{ABD}\to\overline{BC}\to i$

3:
$\tau\to\overline{ab}=\tau\to\widehat{ABD}\to\overline{bc}$ | $\tau\to\overline{ad}=\tau\to\widehat{ABD}\to\overline{ac}$ | $\tau\to\overline{bd}=\tau\to\widehat{ABD}\to\overline{ab}$
$\tau\to\overline{de}=\tau\to\widehat{ABC}\to\overline{ab}$ | $\tau\to\overline{df}=\tau\to\widehat{ABC}\to\overline{ac}$ | $\tau\to\overline{ef}=\tau\to\widehat{ABC}\to\overline{bc}$
$\tau\to\overline{ac}=\tau\to\widehat{ACD}\to\overline{bc}$ | $\tau\to\overline{af}=\tau\to\widehat{ACD}\to\overline{ac}$ | $\tau\to\overline{cf}=\tau\to\widehat{ACD}\to\overline{ab}$
$\tau\to\overline{bc}=\tau\to\widehat{BCD}\to\overline{bc}$ | $\tau\to\overline{be}=\tau\to\widehat{BCD}\to\overline{ac}$ | $\tau\to\overline{ce}=\tau\to\widehat{BCD}\to\overline{ab}$

4:
$\tau\to\widehat{Dab}=\tau\to\widehat{ABD}\to\widehat{Cbc}$ | $\tau\to\widehat{Dac}=\tau\to\widehat{ACD}\to\widehat{Cbc}$
$\tau\to\widehat{Dbc}=\tau\to\widehat{BCD}\to\widehat{Cbc}$ | $\tau\to\widehat{Aaf}=\tau\to\widehat{ACD}\to\widehat{Aac}$
$\tau\to\widehat{Adf}=\tau\to\widehat{ABC}\to\widehat{Aac}$ | $\tau\to\widehat{Aad}=\tau\to\widehat{ABD}\to\widehat{Aac}$
$\tau\to\widehat{Ccf}=\tau\to\widehat{ACD}\to\widehat{Bab}$ | $\tau\to\widehat{Cef}=\tau\to\widehat{ABC}\to\widehat{Cbc}$
$\tau\to\widehat{Cce}=\tau\to\widehat{BCD}\to\widehat{Bab}$ | $\tau\to\widehat{Bbe}=\tau\to\widehat{BCD}\to\widehat{Aac}$
$\tau\to\widehat{Bde}=\tau\to\widehat{ABC}\to\widehat{Bab}$ | $\tau\to\widehat{Bbd}=\tau\to\widehat{ABD}\to\widehat{Bab}$

5: **fin procedimiento**

Algoritmo 3.20 Reorientación para la división del octaedro (*Tipo I*)

1: **procedimiento** ORIENTA_OCTAEDRO(τ)

2: **si** $\min(|\tau\to\overline{bf}|,|\tau\to\overline{cd}|,|\tau\to\overline{ae}|)=|\tau\to\overline{bf}|$ **entonces**

3:
$\tau\to p=\tau\to f$ | $\tau\to q=\tau\to b$ | | |
$\tau\to\overline{pr}=\tau\to\overline{df}$ | $\tau\to\overline{qr}=\tau\to\overline{bd}$ | $\tau\to\overline{ps}=\tau\to\overline{cf}$ | $\tau\to\overline{qs}=\tau\to\overline{bc}$
$\tau\to\overline{pu}=\tau\to\overline{af}$ | $\tau\to\overline{qu}=\tau\to\overline{ab}$ | $\tau\to\overline{pt}=\tau\to\overline{ef}$ | $\tau\to\overline{qt}=\tau\to\overline{be}$

4:
$\tau\to\widehat{prt}=\ \tau\to\widehat{ABC}\to\widehat{abc}$ | $\tau\to\widehat{qrt}=\tau\to\widehat{bde}$
$\tau\to\widehat{pst}=\tau\to\widehat{cef}$ | $\tau\to\widehat{qst}=\ \tau\to\widehat{BCD}\to\widehat{abc}$
$\tau\to\widehat{pru}=\tau\to\widehat{adf}$ | $\tau\to\widehat{qru}=\ \tau\to\widehat{ABD}\to\widehat{abc}$
$\tau\to\widehat{psu}=\ \tau\to\widehat{ACD}\to\widehat{abc}$ | $\tau\to\widehat{qsu}=\tau\to\widehat{abc}$

5: **si no si** $\min(|\tau\to\overline{bf}|,|\tau\to\overline{cd}|,|\tau\to\overline{ae}|)=|\tau\to\overline{cd}|$ **entonces**

6:
$\tau\to p=\tau\to d$ | $\tau\to q=\tau\to c$ | | |
$\tau\to\overline{pr}=\tau\to\overline{df}$ | $\tau\to\overline{qr}=\tau\to\overline{cf}$ | $\tau\to\overline{ps}=\tau\to\overline{bd}$ | $\tau\to\overline{qs}=\tau\to\overline{bc}$
$\tau\to\overline{pu}=\tau\to\overline{de}$ | $\tau\to\overline{qu}=\tau\to\overline{ce}$ | $\tau\to\overline{pt}=\tau\to\overline{ad}$ | $\tau\to\overline{qt}=\tau\to\overline{ac}$

7:
$\tau\to\widehat{prt}=\tau\to\widehat{adf}$ | $\tau\to\widehat{qrt}=\ \tau\to\widehat{ACD}\to\widehat{abc}$
$\tau\to\widehat{pst}=\ \tau\to\widehat{ABD}\to\widehat{abc}$ | $\tau\to\widehat{qst}=\tau\to\widehat{abc}$
$\tau\to\widehat{pru}=\ \tau\to\widehat{ABC}\to\widehat{abc}$ | $\tau\to\widehat{qru}=\tau\to\widehat{cef}$
$\tau\to\widehat{psu}=\tau\to\widehat{bde}$ | $\tau\to\widehat{qsu}=\ \tau\to\widehat{BCD}\to\widehat{abc}$

8: **si no**

9:
$\tau\to p=\tau\to e$ | $\tau\to q=\tau\to a$ | | |
$\tau\to\overline{pr}=\tau\to\overline{de}$ | $\tau\to\overline{qr}=\tau\to\overline{ad}$ | $\tau\to\overline{ps}=\tau\to\overline{ce}$ | $\tau\to\overline{qs}=\tau\to\overline{ac}$
$\tau\to\overline{pu}=\tau\to\overline{be}$ | $\tau\to\overline{qu}=\tau\to\overline{ab}$ | $\tau\to\overline{pt}=\tau\to\overline{ef}$ | $\tau\to\overline{qt}=\tau\to\overline{af}$

10:
$\tau\to\widehat{prt}=\ \tau\to\widehat{ABC}\to\widehat{abc}$ | $\tau\to\widehat{qrt}=\tau\to\widehat{adf}$
$\tau\to\widehat{pst}=\tau\to\widehat{cef}$ | $\tau\to\widehat{qst}=\ \tau\to\widehat{ACD}\to\widehat{abc}$
$\tau\to\widehat{pru}=\tau\to\widehat{bde}$ | $\tau\to\widehat{qru}=\ \tau\to\widehat{ABD}\to\widehat{abc}$
$\tau\to\widehat{psu}=\ \tau\to\widehat{BCD}\to\widehat{abc}$ | $\tau\to\widehat{qsu}=\tau\to\widehat{abc}$

11: **fin si**

12: **fin procedimiento**

3.5. Proceso de Compactación

Durante los procesos de refinamiento y desrefinamiento es necesario eliminar diferentes tipos de elementos. Durante el refinamiento, al marcar un tetraedro como refinable que había sido dividido en transitorios debe suprimirse sus tetraedros hijos y volverse a dividir. En el proceso de desrefinamiento está aún más claro: se eliminan elementos que no aportan información significativa con respecto a la suministrada por sus padres. La rutina es común a ambos procesos ya que su comportamiento es similar, con la única particularidad de que durante la compactación posterior a un refinamiento no se realizará borrado de nodos.

El proceso que realiza la destrucción de los elementos y rehace las estructuras resultantes de la malla añadiendo los nuevos elementos creados y eliminando los destruidos, es denominado proceso de compactación. Hay por lo tanto dos fases claramente diferenciadas.

Originalmente no se implementó este proceso, sino que las estructuras se iban ajustando dinámicamente. El rendimiento de esta manera no era satisfactorio, por lo que se optó por realizar un posproceso que hiciera esta labor de una sola vez. Gracias al rendimiento de las estructuras *vector* y *list* a la hora de mover datos entre ellas se ha conseguido un gran incremento en la velocidad del proceso global.

Finalmente,

3.5.1. Borrado de elementos

La primera acción del proceso de compactación consistirá en realizar recorridos por las diferentes estructuras de la malla para determinar qué elementos han sido marcados para su eliminación definitiva. Por cuestiones de integridad, en primer lugar se tomarán referencias de los elementos, y al final se procederá a su borrado.

En el algoritmo 3.21 se ve un esquema de esta fase. En cada llamada al módulo Procesar_Lista (líneas 2, 3 y 4) se pasa una referencia a una lista de elementos de la malla T. Este módulo (algoritmo 3.22) realiza dos funciones: rehacer la lista pasada como parámetro y devolver una lista de elementos "padre" de aquellos que hay que eliminar.

El proceso indicado se realizará para aquellos elementos que pueden ser divididos: tetraedros, caras y aristas. Para los nodos se mantendrán las referencias

de los no marcados para eliminar. Una vez completadas las listas de elementos a permanecer, se procederá a recorrer las listas de elementos cuya división debe ser suprimida. La llamada a DEHACER_DIVISION (líneas 12, 15 y 18) provocará el borrado definitivo de todos los hijos de los elementos referenciados, así como de cualquier otro elemento que se haya creado interiormente para realizar la división.

Algoritmo 3.21 Borrado de elementos marcados

```
procedimiento BORRADO_ELEMENTOS(T)
    tetaedros_a_deshacer = PROCESAR_LISTA(T→tetraedros)
    caras_a_deshacer = PROCESAR_LISTA(T→caras)
    aristas_a_deshacer = PROCESAR_LISTA(T→aristas)
    nodos_a_mantener = ∅
    para n_i ∈ T → nodos
        si n_i no marcado para eliminar entonces
            nodos_a_mantener ←push— n_i
        fin si
    fin para
    para τ_i ∈ tetaedros_a_deshacer
        DEHACER_DIVISION(τ_i)
    fin para
    para χ_i ∈ caras_a_deshacer
        DEHACER_DIVISION(χ_i)
    fin para
    para α_i ∈ aristas_a_deshacer
        DEHACER_DIVISION(α_i)
    fin para
    T→nodos = nodos_a_mantener
fin procedimiento
```

A la hora de procesar las listas en el algoritmo 3.22, hay que destacar que se van a mantener referencias a los elementos "padre" de aquellos que habrá que eliminar. Se deberá tener en cuenta dos casos diferentes:

- un elemento que está marcado para que su división interna se suprima (linea 5). Se guarda una referencia al propio elemento.
- un elemento marcado para eliminar (linea 7). En este caso se mantiene una referencia a su elemento "padre".

Finalmente se rehace la lista de elementos que fueron enviados para procesar con aquellos que no sufren ningún tipo de variación junto con los que van a

permanecer aunque su división interna se elimine (linea 13).

Algoritmo 3.22 Proceso de listas de elementos

```
1: función PROCESAR_LISTA(lista_proceso)
2:     lista_a_deshacer = ∅
3:     resto = ∅
4:     para e_i ∈ lista_proceso
5:         si e_i marcado para deshacer división entonces
6:             lista_a_deshacer ⟵push e_i
7:         si no si e_i marcado para eliminar entonces
8:             lista_a_deshacer ⟵push e_i→Padre
9:         si no
10:            resto ⟵push e_i
11:        fin si
12:    fin para
13:    lista_proceso = lista_a_deshacer ⋃ resto
14:    devolver lista_a_deshacer
15: fin función
```

3.5.2. Generación de estructuras

Una vez que se han suprimido los elementos que debían desaparecer de la malla, falta por incorporar todos aquellos que se han generado en los procesos de división.

En la generación de las nuevas listas de elementos van a mantenerse igual aquellos elementos que no han sido divididos, mientras que los que hayan sufrido un proceso de división pasarán a ser sustituidos por sus elementos hijos y resto de elementos internos que se hayan creado.

En el algoritmo 3.23 puede verse un esquema completo. El proceso comienza (línea 2) con el paso de referencias de los almacenamientos principales a unas listas temporales, y el vaciado de los principales. Solo se realiza este proceso con los elementos divisibles (tetraedros, caras y aristas). Los nodos, al no ser divisibles, permanecerán en la nueva reestructuración de la malla.

Para cada lista que haya que procesar (líneas 3, 13 y 22) se va a realizar las mismas tareas. Tras extraer un elemento, se verificará si está dividido. En caso de que no lo esté, se incluirá en la lista definitiva de elementos.

Si estuviera dividido, ese elemento no pasaría a la lista definitiva, y serían sus elementos hijos y los elementos internos los que pasarían a incorporarse a las listas de proceso temporales. Esto se ha realizado así puesto que es posible

encontrarse dos etapas sucesivas de refinamiento, y partiendo de un elemento original llegaríamos a sus elementos "nietos". Con este desarrollo se resolvería esta cuestión.

Algoritmo 3.23 Reconstrucción de la malla

```
1:  procedimiento RECONSTRUIR_MALLA(T)
        lista_aristas_a_procesar = T→aristas   | T→aristas = ∅
2:      lista_caras_a_procesar = T→caras       | T→caras = ∅
        lista_tetraedros_a_procesar = T→nodos  | T→tetraedros = ∅
3:      mientras lista_tetraedros_a_procesar ≠ ∅
4:          τ ←pop lista_tetraedros_a_procesar
5:          si τ dividido entonces
6:              lista_tetraedros_a_procesar ←push (τ→hijos)
7:              lista_caras_a_procesar ←push (τ→nuevas_caras)
8:              lista_aristas_a_procesar ←push (τ→nuevas_aristas)
9:          si no
10:             (T→tetraedros) ←push τ
11:         fin si
12:     fin mientras
13:     mientras lista_caras_a_procesar ≠ ∅
14:         χ ←pop lista_caras_a_procesar
15:         si χ dividida entonces
16:             lista_caras_a_procesar ←push (χ→hijas)
17:             lista_aristas_a_procesar ←push (χ→nuevas_aristas)
18:         si no
19:             (T→caras) ←push χ
20:         fin si
21:     fin mientras
22:     mientras lista_aristas_a_procesar ≠ ∅
23:         α ←pop lista_aristas_a_procesar
24:         si α dividida entonces
25:             lista_aristas_a_procesar ←push (α→hijas)
26:             (T→nodos) ←push (α→nuevo_nodo)
27:         si no
28:             (T→aristas) ←push α
29:         fin si
30:     fin mientras
31: fin procedimiento
```

3.6. Análisis computacional

En esta sección se va a realizar un análisis de la implementación del algoritmo para determinar el coste asociado al mismo.

Partiendo del esquema general (algoritmo 3.1), en el que se esquematiza el proceso en tres partes diferenciadas, se estudiará cada una de ellas.

3.6.1. Proceso de marcado

El proceso de marcado realiza, inicialmente, un recorrido por todos los elementos de la malla. Cada vez que se encuentre un elemento a marcar, añade en la lista de elementos a revisar aquellos que son vecinos por arista.

El mejor caso posible se da en los extremos del refinamiento: cuando no se refina ningún elemento, o cuando se refinan todos (global), puesto que cada tetraedro se revisa sólo una vez.

Cuando hay transitorios en la malla el proceso sufre una parada. Debe estudiarse el caso de que se trate, deshacer la división correspondiente y volver a lanzar el proceso por si cambia el número de marcas en algún tetraedro. Dadas las características del algoritmo en el hecho de que un transitorio nunca puede ser dividido, este proceso solo se realizaría una vez en un tetraedro.

En el estudio de los tipos de marcado hay que indicar una limitación impuesta por el algoritmo: se procesará un tetraedro como máximo cuatro veces. Si está indicado para refinar, en la primera vez que se estudie se marcarán todas sus aristas. En caso de que vaya recibiendo marcas tendrá que volver a ser estudiado, pero en el peor caso, al llegar a cuatro marcas, se le añadirán las dos restantes por criterios de conformidad del algoritmo.

Se puede ver en los párrafos anteriores los peores casos posibles en el procesamiento de tetraedros: que tenga que eliminarse su división interna y que reciba las marcas de una en una. No obstante, no es posible que este caso se dé para todos los tetraedros de una malla (debe haber al menos uno refinable para arrancar el proceso en una hipotética malla totalmente dividida en transitorios proveniente de un desrefinamiento).

En cualquier caso, se está trabajando con número fijos, por lo que estamos ante un algoritmo con un coste lineal, asociado al número de elementos a procesar ($O(n)$).

3.6.2. Proceso de división

En este proceso, de forma general, se realiza un recorrido por todos los elementos de la malla para determinar cuáles deben ser divididos en función del número de marcas que tengan. Puesto que los elementos solo están una vez en la malla, nunca van a ser procesados dos veces, por lo que estamos ante un proceso de tipo $O(n)$.

Los algoritmos de división emplean un gran número de asignaciones, necesarias en los proceso de reorientación de aristas y caras. No obstante, la paralelización de la división de los elementos hace aumentar el rendimiento global.

El número de asignaciones es fijo en función del tipo de división. A medida que crece el número de elementos, lo hacen proporcionalmente las asignaciones. Es un proceso claramente de coste lineal.

3.6.3. Proceso de compactación

En las dos fases del proceso de compactación se hacen recorridos sobre los almacenamientos de la malla. Hay un número fijo de ellos, independiente del estado de los elementos.

El peor caso lo tenemos en la reconstrucción de la malla (algoritmo 3.23), en la que se van añadiendo elementos en las listas de proceso. En el caso de un refinamiento global, para cada elemento dividido se añadirán sus hijo, por lo que el coste total sería:

$$O_r = \sum_{i=n,a,c,t} O(n_i + \rho_i n_i) \qquad \begin{array}{l} n\text{=nodos} \\ a\text{=aristas} \\ c\text{=caras} \\ t\text{=tetraedros} \end{array} \tag{3.2}$$

n_i representa el número de nodos ($i = n$), aristas ($i = 1$), caras ($i = c$) y tetraedros ($i = t$) antes de la división y ρ_i sería el número máximo de elementos (nodos, aristas, caras y tetraedros) en los que puede dividirse uno determinado ($\rho_n = 0$, $\rho_a = 2$, $\rho_c = 4$ y $\rho_t = 8$).

Incluso en el peor caso mostrado, el orden asociado es siempre lineal, en función del número de elementos de la malla y del número de divisiones realizadas.

3.6.4. Proceso global

Tras analizar los tres puntos básicos del algoritmo, y en cada uno de ellos llegar a la conclusión de que son de tipo $O(n)$, el proceso global debe considerarse de la misma manera.

El proceso de división, a pesar de su sencillez conceptual, es el que consume mayor tiempo de ejecución, con diferencia, frente a las otras dos fases. La paralelización del proceso ha mitigado esta diferencia que originalmente fue enorme.

Capítulo 4

Algoritmo de Desrefinamiento

4.1. Presentación

Se puede definir el algoritmo de desrefinamiento como el inverso del algoritmo de refinamiento anteriormente presentado [González-Yuste et al., 2004a]. La idea es eliminar elementos desde la malla más fina hasta la más gruesa atendiendo a su solución numérica.

Existen diferentes variantes en función de qué tipo de elementos se consideren para establecer el criterio de desrefinamiento. En este caso se van a emplear los nodos. No obstante, hay que indicar que el hecho de emplear otro elemento (aristas, caras o tetraedros) únicamente variará la aplicación del criterio, puesto que el estudio de qué elementos serán finalmente eliminados dependerá de los criterios de conformidad que establece el algoritmo de refinamiento.

En el algoritmo de refinamiento propuesto, un nodo $n_p^{j-1} \in M_{j-1}$ permanecerá en la malla M_j como n_p^j. Si en la malla M_{j-1} hubiera una arista formada por los nodos n_p^{j-1} y n_q^{j-1}, y esta arista fuera refinada por un nodo n_i^j, en la malla M_j tendríamos una arista formadas por n_p^j y n_i^j, y otra por los nodos n_q^j y n_i^j.

Una vez resuelto el problema asociado, todos los nodos de M_j tendrán una solución numérica v^j. Para determinar si el nodo n_i^j puede ser eliminado de la malla M_j se comparará su solución numérica con la interpolada de los nodos que pertenecen a la arista que divide. Formalmente:

$$\left| v_i^j - \frac{v_p^j + v_q^j}{2} \right| < \epsilon \tag{4.1}$$

En definitiva, el parámetro de desrefinamiento ϵ indica la precisión que se desea alcanzar para la solución numérica. Este criterio de desrefinamiento es la extensión inmediata a 3D de la condición introducida en [Ferragut et al., 1994] para la aproximación de funciones de dos variables.

Existen, además, una serie de criterios que se deben tener en cuenta a la hora de determinar si un elemento puede ser suprimido de la malla:

- Los elementos de la malla base (M_0) no pueden eliminarse. Estos elementos deben permanecer siempre.
- Un elemento no puede ser eliminado si está dividido. Dicho de otra manera, si un elemento debe permanecer en la malla, todos sus "antecesores" también lo harán.
- Si un elemento permanece en la malla, deberán permanecer todos los elementos que lo contienen.
- Deberán permanecer en la malla todos los elementos necesarios para cumplir los criterios de conformidad que establece el algoritmo de refinamiento.

Como puede observarse, la definición de este algoritmo es muy sencilla, si bien, se están asumiendo todos los criterios previamente definidos en el refinamiento, por lo que no es necesario duplicarlos en este punto.

4.2. Implementación

Al igual que en el caso del refinamiento, este algoritmo tiene una serie de fases bien diferenciadas:

- Marcado
- Revisión
- Conformado

Inicialmente, se deben determinar qué elementos pueden ser eliminados por el criterio de desrefinamiento. Además, se elaborarán una serie de listas de tetraedros por nivel de profundidad, necesarias para el proceso de revisión.

Para cada uno de los niveles de profundidad de la malla será necesario realizar un estudio de qué elementos, atendiendo a los criterios de conformidad,

deben permanecer. Una vez completado cada nivel, se puede realizar el borrado de los elementos seleccionados.

Finalmente, se realizará un proceso de división de los elementos que hayan quedado marcados para disponer de una malla final conforme a las especificaciones de los algoritmos.

4.2.1. Aplicación a otros elementos

La implementación del algoritmo está totalmente orientada al tratamiento de los nodos, tanto en la fase de marcado por el criterio de desrefinamiento, como en el módulo de revisión.

No obstante, es perfectamente posible realizar un tratamiento de otro tipo de elemento en la malla para realizar el proceso de marcado. Se pueden realizar, con el criterio adecuado, revisiones sobre aristas, caras o tetraedros sin alterar la programación del desrefinamiento.

La única condición es que se marquen los nodos asociados a los elementos que se pretendan estudiar en el caso necesario. De esta manera encajaría perfectamente en el resto de los algoritmos.

Incluso, el criterio de desrefinamiento (4.1) puede generalizarse a cuestiones que no estén relacionadas directamente con la solución numérica obtenida por el MEF sobre la malla M_j. Por ejemplo, se puede pensar en criterios basados en aspectos puramente geométricos o de aproximaciones de funciones definidas en la malla de tetraeros.

4.3. Proceso de marcado

Para la implementación del marcado se va a seguir un esquema inverso al planteado de modo conceptual. El algoritmo 4.1 muestra el proceso genérico sobre una malla T.

Inicialmente, todos los nodos de la malla (excepto los de la malla base) serán marcados como desrefinables. Un indicador asociado a cada nodo va a determinar aquellos que son “candidatos” a ser eliminados. El resto de procesos irá encaminado a determinar cuáles deben permanecer.

El siguiente paso será evaluar la condición de desrefinamiento (línea 7). Se ha colocado esta rutina como un módulo aparte, puesto que puede sustituirse por cualquier otro que cumpla una función similar empleando otro criterio.

Algoritmo 4.1 Proceso de marcado

```
1: procedimiento MARCADO(T)
2:     para n_i ∈ T                                   ▷ Marcar todos
3:         si n_i→Nivel > 0 entonces                  ▷ No tocar nivel cero
4:             n_i→Desrefinable = true
5:         fin si
6:     fin para
7:     EVALUAR_CONDICION_DESREFINAMIENTO(T)
8: fin procedimiento
```

La forma implementada para evaluar la condición de desrefinamiento (algoritmo 4.2) recorrerá la malla y computará dicha evaluación en todos los nodos. Aquellos nodos que no satisfagan dicha condición deberán ser desmarcados. Para realizar esta última acción se va a invocar un proceso recursivo de propagación para el desmarcado (línea 5).

Algoritmo 4.2 Evaluación de la condición de desrefinamiento

```
1: procedimiento EVALUAR_CONDICION_DESREFINAMIENTO(T)
2:     para n_i ∈ T
3:         si n_i→Nivel > 0 entonces                  ▷ No evaluar nivel cero
4:             si |v_i − (v_p+v_q)/2| ≥ ε entonces
5:                 PROPAGAR_NO_DESREFINABLE(n_i)
6:             fin si
7:         fin si
8:     fin para
9: fin procedimiento
```

Cada vez que un nodo es desmarcado (es decir, deja de ser "candidato" a ser eliminado) se lanza un proceso recursivo (algoritmo 4.3). Puesto que si un elemento debe permanecer en la malla también lo harán aquellos que lo contienen, al mantener un nodo en la malla deberá permanecer, al menos, la arista que divide. Y si debe permanecer dicha arista, lo harán las caras y tetraedros que la contienen. Por lo tanto, desmarcando los nodos que forman los tetraedros que contienen la mencionada arista, se conseguirá este efecto.

Para cada nuevo nodo desmarcado se volverá a lanzar este proceso, para efectuar esta operación hacia niveles de malla más gruesa. El necesario criterio de parada se establecerá cuando se encuentre un nodo que no tenga la marca de desrefinamiento, bien sea porque ya ha sido procesado o porque pertenezca al nivel cero.

Para identificar la arista que es dividida por un nodo n, se empleará la notación α_n.

Algoritmo 4.3 Propagación de la condición de no-desrefinable

```
1: procedimiento PROPAGAR_NO_DESREFINABLE(n_i)
2:     si not n_i→Desrefinable entonces                ▷ criterio de parada
3:         fin
4:     fin si
5:     n_i→Desrefinable = false
6:     para τ_i ∈ Ω_{α_{n_i}}                          ▷ tetraedros formados por α_{n_i}
7:         para n_j ∈ N_{τ_i}                          ▷ procesar cada nodo del tetraedro
8:             PROPAGAR_NO_DESREFINABLE(n_j)
9:         fin para
10:    fin para
11: fin procedimiento
```

4.4. Proceso de revisión

Una vez completado el proceso de marcado se va a disponer de una malla con la información de qué nodos pueden ser eliminados y cuáles no atendiendo al criterio de desrefinamiento.

En este punto ya se podría afrontar la eliminación de los elementos que pueden ser eliminados, pero sería posible encontrarse elementos que no cumplan los criterios de conformidad de la malla, es decir, que tuvieran un número de nodos no marcados (a permanecer) inválido a efectos del algoritmo de refinamiento, lo que provocaría una nueva introducción de nodos y que vuelvan a ser divididos como estaban.

El proceso de revisión va a suprimir marcas de nodos desrefinables atendiendo a los criterios de conformidad del refinamiento. Una segunda fase realizará el marcado para que sean posteriormente eliminados aquellos elementos que definitivamente puedan desaparecer.

Para abordar estos procesos se ha diseñado un esquema de trabajo por niveles de malla. Partiendo desde los niveles más finos de la malla, se van a procesar los tetraedros según su nivel. El esquema general se presenta en el algoritmo 4.4.

Algoritmo 4.4 Esquema general del proceso de revisión

```
1: procedimiento REVISION(T)
2:     listas_por_nivel= ∅
3:     profundidad = GENERAR_LISTAS(T,listas_por_nivel)
4:     mientras profundidad ≥ 0
5:         PROCESAR_LISTA_NIVEL(listas_por_nivel[profundidad])
6:         DESHACER_NIVEL(T, listas_por_nivel[profundidad])
7:         profundidad = profundidad - 1
8:     fin mientras
9: fin procedimiento
```

4.4.1. Generación de listas por niveles

Es este subproceso se creará una serie de listas de tetraedros a estudiar por cada nivel de profundidad de la malla. Partiendo de los tetraedros de la malla se insertarán los padres de dichos tetraedros en cada lista.

Posteriormente se estudiarán estos tetraedros padres ya que son los que podrán ver suprimidas sus divisiones internas. Posteriormente, si a un tetraedro se le elimina su división interna, el padre del tetraedro será añadido en la lista correspondiente. Esto se haría en el subproceso 4.4.3.

El algoritmo 4.5 muestra un esquema de la generación mencionada. Las referencias de los tetraedros padres de los tetraedros que conforman la malla fina se guardan en una estructura tipo array de listas, en la que cada posición del array se corresponde con una lista de tetraedros pertenecientes a un nivel.

Una segunda utilidad del proceso es determinar la profundidad máxima de la malla, cosa que se hace mientras se recorren los elementos para ser procesados (línea 6).

Algoritmo 4.5 Generación de listas de tetraedros por niveles

```
1: función GENERAR_LISTAS(T,listas_por_nivel)
2:     max_nivel = 0
3:     para τ_i ∈ T
4:         si τ_i→Nivel>0 entonces
5:             listas_por_nivel[τ_i→Padre→Nivel] ⟵push τ_i→Padre
6:             max_nivel = máx(max_nivel,τ_i→Padre→Nivel)
7:         fin si
8:     fin para
9:     devolver max_nivel
10: fin función
```

4.4.2. Procesamiento de listas de un nivel

Una vez completadas las listas por niveles se deberá estudiar cada nivel. El algoritmo 4.6 muestra el esquema de revisión de una lista de un nivel cualquiera. El proceso consiste en determinar qué tetraedros no serán desrefinados debido a los criterios de conformidad del algoritmo de refinamiento.

Algoritmo 4.6 Esquema del proceso de revisión

```
1:  procedimiento PROCESAR_LISTA_NIVEL(lista_de_nivel)
2:      repetir
3:          lista_cambiada = false
4:          lista_admitidos = ∅
5:          para τ_i ∈lista_de_nivel
6:              si τ_i→Hijos = ∅ entonces        ▷ sin hijos, no se tiene en cuenta
7:                  lista_cambiada = true
8:                  si τ_i→Padre→Nivel>0 entonces          ▷ padre a estudiar
9:                      listas_por_nivel[τ_i→Padre→Nivel] ⟵push τ_i→Padre
10:                 fin si
11:                 continuar bucle "para"
12:             fin si
13:             tetraedro_valido = true
14:             si τ_i→Hijos = 8 entonces          ▷ si dividido en 8, estudiar
15:                 si A*_τi ∈ [1, 2, 3_=cara] entonces          ▷ criterio inverso
16:                     tetraedro_valido = false
17:                 fin si
18:             si no si A*_τi = ∅ entonces          ▷ ninguna arista marcada
19:                 tetraedro_valido = false
20:             fin si
21:             si tetraedro_valido entonces
22:                 lista_admitidos ⟵push τ_i          ▷ tetraedro se mantiene
23:             si no
24:                 lista_cambiada = true
25:                 para α_j ∈ A*_τi          ▷ mantenemos su división
26:                     PROPAGAR_NO_DESREFINABLE(α_j→i)
27:                 fin para
28:             fin si
29:         fin para
30:         lista_de_nivel= lista_admitidos
31:     hasta not lista_cambiada
32: fin procedimiento
```

Se va a emplear una lista auxiliar (línea 4) y una variable de control (línea 3). Realizando sucesivas pasadas sobre la lista del nivel, se van a incluir en la

lista auxiliar aquellos tetraedros que mantendrán su división (línea 22). Si un tetraedro no se incluye, la variable de control tomará un valor **true**. Al finalizar el bucle sobre la lista de tetraedros, se generará una nueva lista de nivel (línea 30), y si la variable de control indica que ha habido cambios sobre la lista, se volverá a iniciar el proceso.

Para cada tetraedro que se procese se realizarán varias comprobaciones mutuamente excluyentes:

- No está dividido (línea 6). Estos tetraedros no se podrán desrefinar, no se pasarán a la lista auxiliar y serán descartados. Su padre deberá ser incluido en la lista del nivel inmediatamente anterior para que se estudie en el futuro.

- Está dividido en como *Tipo I* (línea 14). Se debe mirar el número de aristas marcadas para desrefinar. El criterio es justo el inverso empleado en el algoritmo 3.3 para marcar tetraedros por conformidad. En caso de que cumpla el criterio, se podrá mantener como desrefinable. Si no, indica que el tetraedro volverá a recibir los nodos que se eliminen por conformidad, por lo que no tiene sentido deshacer su división, y se activará un indicador de tetraedro inválido.

- No tiene aristas marcadas (línea 18). Debido a que sus descendientes han ido desmarcando nodos, es posible que haya tetraedros sin aristas marcadas para desrefinar. Es este caso, el tetraedro no se podrá desrefinar, activando el mismo indicador del caso anterior.

Si supera las tres comprobaciones anteriores tendremos un tetraedro dividido en transitorios y con aristas marcadas. Estos tetraedros siempre se podrán desrefinar ya que habrá otra división posible con un menos número de aristas marcadas. Será incluido en la lista auxiliar.

En caso de que el tetraedro se considere no válido para desrefinar, se desmarcarán aquellos nodos que estuvieran marcados (línea 26). Esto podría provocar que tetraedros de este mismo nivel ya no fueran declarados válidos, por lo que sería necesario revisar de nuevo los tetraedros que permanezcan en la lista auxiliar.

El operador A^*_τ representa la listas de aristas marcadas para desrefinar.

4.4.3. Eliminar elementos de un nivel

Una vez se ha determinado qué tetraedros pueden ser desrefinados dentro de un nivel, se procederá a eliminar sus divisiones internas, así como todas las caras y aristas introducidas y generadas para dividir los elementos de ese nivel. El algoritmo 4.7 muestra el proceso.

Algoritmo 4.7 Esquema del proceso de eliminación de un nivel

```
1: procedimiento DESHACER_NIVEL(T, lista_de_nivel)
2:     lista_caras = ∅
3:     para τ_i ∈ lista_de_nivel
4:         si τ_i→Nivel>0 entonces          ▷ si tiene padre se pone en estudio
5:             listas_por_nivel[τ_i→Padre→Nivel] ←push τ_i→Padre
6:         fin si
7:         Marcar τ_i para deshacer división
8:         Desmarcar τ_i como refinable
9:         lista_caras ←push C_τi
10:    fin para
11:    lista_aristas = ∅
12:    para χ_i ∈ lista_caras
13:        si χ_i → Hijos ≠ ∅ y A*_χi ≠ ∅ entonces
14:            Marcar χ_i para deshacer división
15:            para α_j ∈ A_χi
16:                si α_j → Nivel = χ_i → Nivel entonces
17:                    lista_aristas ←push α_j
18:                fin si
19:            fin para
20:        fin si
21:    fin para
22:    para α_k ∈ lista_aristas
23:        si α_k → Hijos ≠ ∅ y α_k → i marcado como desrefinable entonces
24:            Marcar α_k para deshacer división
25:            Desmarcar(α_k)
26:        fin si
27:    fin para
28:    BORRADO_ELEMENTOS(T)
29: fin procedimiento
```

Partiendo de la lista de tetraedros desrefinables, se añadirá el padre de cada uno de ellos en la lista correspondiente para su estudio posterior (línea 5). A continuación se marcará para que se deshaga su división interna, se elimina la marca de refinable para evitar que sea dividido como *Tipo I* en posteriores

procesos y, finalmente, se añaden todas sus caras en una lista de caras a estudiar (línea 9).

Con la lista de caras obtenida en el proceso anterior se va a realizar un recorrido para determinar si deben ser desrefinadas. Siempre que estén divididas y tengan alguna arista desrefinable (línea 13), se marcará la cara para deshacer su división interna y para cada una de sus aristas se deberá comprobar si pertenecen al mismo nivel que la cara (que no hayan sido introducidas en niveles anteriores). En caso afirmativo, la arista se añadirá en una lista de aristas a estudiar (línea 17).

Finalmente, para las aristas seleccionadas en el paso anterior se realizará un proceso similar: se mirará si está dividida y el nodo que la divide está marcado como desrefinable (línea 23) y en caso de que así sea, se marcará para deshacer su división y se elimina la marca de arista a dividir para evitar que lo sea en futuros procesos.

En todo el proceso únicamente se han realizado marcas para que se eliminen las divisiones internas de los elementos. La división efectiva se hará en el módulo Borrado_Elementos (línea 28), que realiza una llamada al procedimiento descrito en la sección 3.5.1.

4.5. Conformado

La malla resultante de los procesos anteriores tendrá una serie de tetraedros sin divisiones internas pero con aristas marcadas y divididas. Esto es una malla no conforme. A nivel computacional, tras la eliminación de las divisiones internas, las estructuras de datos están cargadas de referencias inexistentes, por lo que deberían ser suprimidas.

El módulo de conformado realizará dos tareas conocidas:

1. Compactar las estructuras. En la sección 3.5.2 se presentó el algoritmo 3.23 para adecuar las listas de elementos a la malla resultante. Se invocará este módulo para obtener el resultado deseado.

2. Conformar la malla. Hay que realizar las divisiones internas de los elementos que han quedado en la malla. Se llamará al módulo completo de refinado (sección 3.2) para que se efectúen las divisiones. Aunque el proceso de refinamiento añade marcas, en este caso no será necesario, puesto

que la malla de trabajo no tiene tetraedros marcados por el indicador de error, y las aristas ya están marcadas y divididas ya que tienen nodos que no se han desrefinado. La tarea principal es la división de los elementos.

La malla final resultante es conforme y queda adaptada a la solución numérica según el parámetro de desrefinamiento indicado.

4.6. Análisis computacional

La implementación del algoritmo de desrefinamiento es bastante más sencilla que la del refinamiento. Está basada, fundamentalmente, en recorrido de listas para el procesado de elementos. Estos recorridos presentan un coste computacional mínimo, por lo que no suponen una carga para el algoritmo.

Podemos encontrar dos puntos que no siguen esta tónica: la propagación entre nodos de la condición de no desrefinable y el estudio de tetraedros por nivel.

En la propagación de la condición de no desrefinable se produce una serie de llamadas para cada nodo que forma un tetraedro que contiene la arista dividida por el nodo que originó la llamada. Efecto dominó se produce en cascada hacia los niveles más gruesos de la malla, por lo que el número de llamadas recursivas podría ser alto.

Sin embargo, cada nodo solo debe ser procesado una vez. Cuando un nodo pierde la condición de desrefinable, no la vuelve a recuperar. Uno de los criterios de parada del proceso recursivos es que el nodo haya sido procesado anteriormente. Si es así, finaliza el proceso en ese punto, y no se generan más llamadas.

Considerando una sola llamada a este proceso se podría pensar que tiene un coste exponencial, pero considerándolo de forma global se puede deducir que tiene un coste lineal, ya que se realiza un solo estudio por nodo.

En el estudio de tetraedros por nivel la situación es diferente. Se parte de una lista inicial con todos los tetraedros del nivel. Se estudia cada uno y si no debe ser desrefinado, se quita de la lista y se marcan como no desrefinable todo sus nodos. Esto provoca que otros tetraedros del mismo nivel, previamente procesados perdieran su condición de desrefinable, por lo que habría que estudiar de nuevo la lista de los elementos restantes.

En la primera revisión de la lista se va a realizar el mayor trabajo de este proceso. Es cuando se eliminarán el mayor número de tetraedros porque habrá nodos no marcados como desrefinables bien por el criterio o porque los tetraedros hijos de los del presente nivel los han desmarcado de forma recursiva. El resto de pasadas sobre la lista de tetraedros restantes únicamente irá ajustando dicho nivel, eliminando de la lista aquellos no conformes, pero no hará grandes modificaciones.

Es imposible determinar a priori el número de pasadas que hay que realizar sobre la lista de un nivel. Suponiendo un caso improbable de una lista con n elementos en las que en cada pasada se eliminaran unos pocos elementos, estaríamos ante un caso de algoritmo de coste $O(n^2)$.

Sin embargo, hay varias cuestiones que mitigan este efecto. La primera ya reseñada es que en la primera pasada se realiza la mayor supresión de elementos de la lista. La segunda es que en cada lista se trabaja con un conjunto de tetraedros, decreciente en número en niveles gruesos de malla, lo que acelera el coste global.

En las implementaciones que se han realizado el tiempo de cómputo del proceso de desrefinamiento es menor que el del refinamiento. En el caso de los procesos de revisión de listas se ha observado que en la primera pasada de cada lista se elimina más de 80 %-90 % del total de elementos que se eliminarán de esa lista. No más de diez pasadas suelen ser más que suficientes para completar el proceso, lo cual es próximo al resultado de [Löhner y Baum, 1992] en un proceso de conformado similar implementado en un algoritmo de refinamiento.

Capítulo 5

Aplicaciones

5.1. Mallas de triángulos en 3D

5.1.1. Introducción

Las mallas de triángulos en tres dimensiones proporcionan una descripción de una geometría mucho más simple, para ciertas aplicaciones, que las ofrecidas por mallas de tetraedros. Mientras que en una malla de tetraedros se realiza la digitalización de todo el volumen de un dominio, las mallas de triángulos sólo presentan información de la superficie.

En ciertos tipos de aplicaciones no es necesario conocer más que la superficie. Existen bastantes aplicaciones que realizan mallados de triángulos en 3D y en internet hay disponible una cantidad amplia de mallas de este tipo.

Puesto que los algoritmos que se presentan trabajan con objetos en tres dimensiones, se podrían considerar las mallas de triángulos como un subconjunto de las mallas de tetraedros. Se ha realizado una fácil adaptación de los desarrollos hechos para poder trabajar con mallas de triángulos y abarcar un espectro mucho mayor de aplicaciones.

El INRIA (*Institut National de Recherche en Informatique et en Automatique*, http://www.inria.fr) ha recopilado más de treinta mil mallas de triángulos que pone a libre disposición, siempre con fines investigadores. Se han escogido dos como muestra de la adaptación del algoritmo de refinamiento.

5.1.2. Adaptación del algoritmo de refinamiento

El objetivo es realizar el mínimo número de modificaciones, y que fueran lo más simple posible, en los algoritmos implementados para permitir su uso en el tipo de malla ya mencionado.

En general, la consideración principal es que se va a trabajar en mallas sin tetraedros. Las mallas de tetraedros y de triángulos tienen en común los nodos, aristas y caras. Por lo tanto, se deberá tener en cuenta que estas mallas tienen un nivel de trabajo menor que para las que originalmente se programaron los módulos.

5.1.2.1. Lectura de ficheros

Hay un formato ampliamente utilizado para definir mallas de triángulos que es ligeramente diferente a los ya implementado. Los fichero `.mesh` presentan información de nodos y triángulos por nodos.

La adaptación del formato de entrada pasa por generar las aristas a partir de los triángulos y qué nodos los componen. De esta manera se consigue homogeneizar el formato `.mesh` con el empleado por los algoritmos.

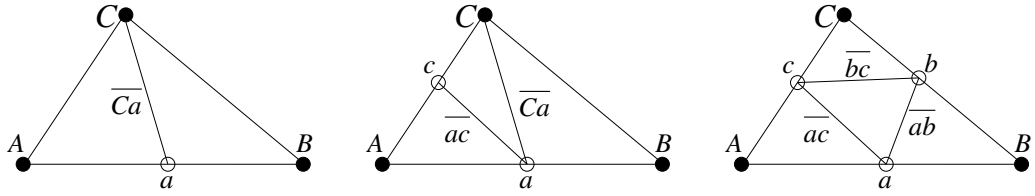
Dado que no existen tetraedros, la lista de tetraedros leídos del fichero deberá quedar vacía. Pero esto no ha supuesto ninguna modificación, ya que el sistema de carga está basado en listas, y al encontrarse una vacía no realiza ninguna acción.

5.1.2.2. Refinamiento

El algoritmo original de refinamiento está orientado al estudio de tetraedros. El proceso de marcado se realiza en las aristas y se hace el estudio de conformidad de acuerdo al número de marcas. El proceso de división determina, en función del número y posición de las marcas, la división a realizar.

Para poder aplicar el mismo sistema en las mallas de triángulos, se debe adaptar los proceso de marcado, conformado y división de elementos.

En primer lugar, los modelos de división serán los mismos que se han empleado en las división de caras del algoritmo original. La figura 5.1 muestra los tres tipos posibles.



(a) 1 marca (b) 2 marcas (c) 3 marcas

Figura 5.1: Formas de dividir una cara según el número de marcas

En el modelo de datos asociado a las caras se ha incluido un indicador de refinable, una marca para señalar cuando una cara debe ser dividida en cuatro. A partir de aquí, el recorrido para realizar marcas es muy simple, puesto que no es necesario realizar estudios de conformidad ya que sólo hay dos tipos de división en función del número de marcas. Es decir, en las mallas de triángulos no hay efecto de propagación.

Lo que sí se tiene en cuenta son las caras "transitorias". Son aquellas resultantes de la división mostrada en las figuras 5.1(a) y 5.1(b). En los casos en que una cara transitoria tuviera que ser refinada bien por su indicador o porque ha recibido una marca en una arista propia se procederá de forma similar que con los tetraedros: se deshace su división interna y se marca como refinable. El esquema completo se puede ver en el algoritmo 5.1.

El algoritmo presenta una diferencia sustancial con respecto al de los tetraedros: se realiza todo dentro de un único bucle, recorriendo una lista de caras a las que se le van añadiendo elementos a medida que se realizan nuevas marcas. Esto ha sido posible, fundamentalmente, y como se ha mencionado, a que no se produce efecto "dominó" en las mallas, lo que permite disminuir el número de casos así como su complejidad.

Dentro del algoritmo se han definido tres posibles acciones:

- Deshacer (línea 31). Esta operación eliminará la división interna del padre de la cara en estudio. Se realizará invocando al procedimiento descrito en el algoritmo 3.5, que marcará los elementos para que posteriormente sean suprimidos. Una vez marcados para su supresión no deberán ser procesados, por lo que se ha incluido una condición especial al inicio de las verificaciones (línea 5) para evitar esta circunstancia.

- Marcar (línea 34). Se realiza cuando una cara debe ser refinada completamente. Se recorren las aristas no marcadas de la cara, se realiza el proceso

Algoritmo 5.1 Refinamiento de mallas de triángulos

```
1: función REFINAR_MALLA_TRIANGULOS(T_k)
2:     lista_de_estudio= C_{T_k}
3:     mientras lista_de_estudio≠ ∅
4:         χ ←pop lista_de_estudio
5:         si χ marcada para borrar entonces
6:             continuar mientras
7:         fin si
8:         deshacer = false
9:         marcar = false
10:        dividir = false
11:        si χ refinable entonces
12:            si χ transitoria entonces
13:                deshacer = true
14:                marcar = true
15:                dividir = true
16:            si no
17:                marcar = true
18:            fin si
19:        si no
20:            si χ marcada entonces
21:                si A*_χ ⊂ A_{χ→Padre} entonces
22:                    deshacer = true
23:                    lista_de_estudio ←push χ→Padre
24:                si no
25:                    deshacer = true
26:                    marcar = true
27:                    dividir = true
28:                fin si
29:            fin si
30:        fin si
31:        si deshacer entonces
32:            DESHACER_TRANSITORIOS_CARA(χ→Padre)
33:        fin si
34:        si marcar entonces
35:            para α_i ∈ A'_χ
36:                Marcar(α_i)
37:                lista_de_estudio ←push X_{α_i}
38:            fin para
39:        fin si
40:        si dividir entonces
41:            DIVIDIR(χ→Padre)
42:            lista_de_estudio ←push χ→Padre→Hijos
43:        fin si
44:    fin mientras
45: fin función
```

de marcado y se añaden a la lista de caras la que se haya visto afectada (sólo habrá una vecina) por esta situación para que sea posteriormente estudiada.

- Dividir (línea 40). Se procede a la división del tetraedro padre del estudiado. Se emplea los módulos descritos en la sección 3.4.8. Una vez dividido se incorporarán las nuevas caras creadas a la lista de caras a estudiar.

El uso de alguna de estas acciones dependerá de los casos de estudio en los que se pudiera encontrar una cara cualquiera:

- Si estuviera indicada para refinar y no fuera transitoria (línea 16), se realizará una operación de marcado de aristas.

- Si estuviera indicada para refinar y además fuera transitoria (línea 12), habría que eliminar la división interna del padre, dividir dicho padre y pasar al estudio de los hijos.

- Una cara ya marcada en unas aristas que pertenecen a la cara padre (línea 21). Sería suficiente con deshacer la división interna del padre de la actual, y añadir el padre a la lista de estudio. De esta manera, el módulo principal de división volvería a dividir al padre en función de nuevo número de marcas.

- Similar al anterior, pero con alguna arista marcada que no pertenece a su padre (línea 24). Este caso se revuelve similar al del segundo punto, teniendo que actuar sobre la cara padre de la cara en estudio.

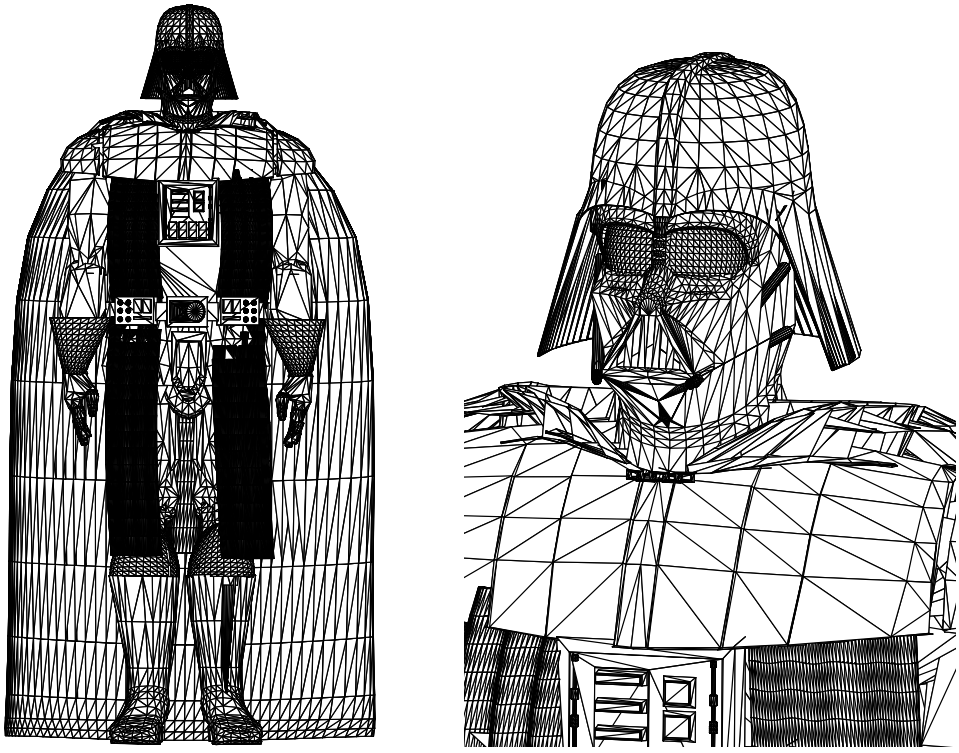
Como automatización del proceso, y para permitir que las aplicaciones funcionen de forma independiente al tipo de malla, se realiza una comprobación previa: si la malla no tiene tetraedros se considerará como de triángulos. En caso de que hubiera tetraedros, se refinará normalmente.

Una vez concluido este proceso de realizaría una llamada al módulo de división (algoritmo 3.7), que aunque tiene una parte dedicada a tetraedros, al no estar presentes en la malla, no realizaría tareas.

Finalmente una llamada a los métodos de compactación y borrado, los mismos descritos en la sección 3.5, dejará las estructuras de datos perfectamente ajustadas.

5.1.3. Aplicaciones

Desde el INRIA se ha descargado una malla sobre las que han realizado unos test. Se corresponde con la figura de Darth Vader (personaje de la saga Star-Wars). Consiste en una malla de 23661 nodos y 45786 triángulos (figura 5.2).



(a) Malla completa (b) Detalle del torso y cabeza

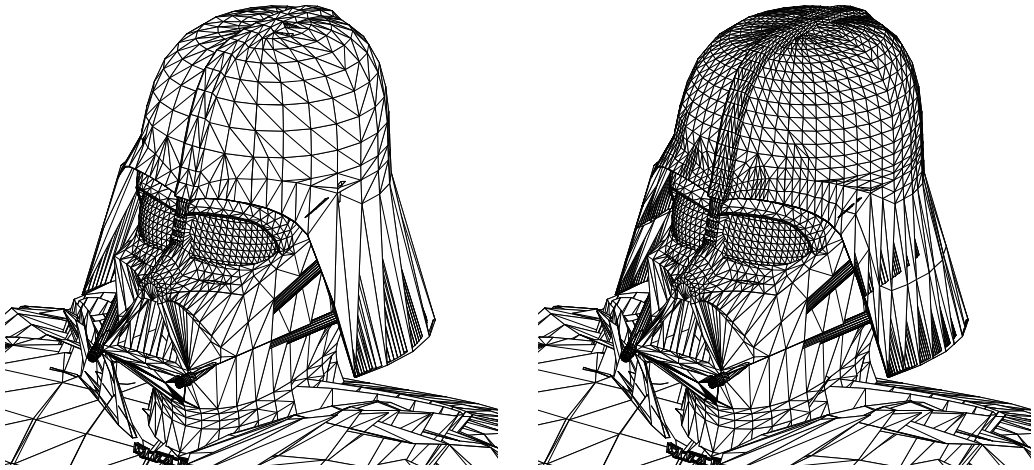
Figura 5.2: Mallas iniciales de Darth Vader (obtenidas de INRIA)

El test que se ha realizado consiste en realizar un refinamiento de la zona del casco de la figura (figura 5.3). La malla pasa a tener 45589 nodos y 89178 tetraedros, generados en un tiempo de CPU inferior a unos pocos segundos. Se ha hecho para comprobar la validez de las modificaciones realizadas.

5.2. Suavizado de mallas en 3D

5.2.1. Introducción

En el método de elementos finitos la calidad de la malla es un aspecto fundamental para el buen comportamiento numérico del método. Algunos generadores



(a) Original (b) Refinada

Figura 5.3: Refinamiento de la cabeza (Darth Vader - INRIA)

automáticos de mallas en 3D construyen mallas de calidad no suficientemente buena y en ocasiones, cuando por diversas razones, se modifican las posiciones de los nodos, pueden aparecer elementos invertidos. Es necesatio disponer de algoritmos capaces de optimizar una malla. Y no sólo de mejorar su calidad (suavizado), sino la de corregir los elementos invertidos (desenredo).

Hay dos formas básicas de mejorar la calidad de una malla. La primera, habitualmente llamada optimización de mallas, consiste en mover cada nodo a una nueva posición que incremente la calidad de los elementos que lo comparten. Esta técnica conserva la topología de la malla, es decir, no modifica la conectividad de los nodos. La segunda técnica implica cambios en la conexión de los nodos. Por ejemplo, el *edge swapping* es una técnica que se podría incluir en esta última categoría.

En [Escobar et al., 2005] se propone un método de optimización que se puede aplicar directamente a mallas con elementos invertidos, haciendo que sea innecesario el proceso de desenredo previo [Montenegro et al., 2002b; Escobar et al., 2003]. Este proceso simultáneo permite reducir el número de iteraciones para alcanzar una calidad determinada en relación a otras estrategias [Knupp, 2001, 2000; Freitag y Knupp, 2002; Freitag y Plassmann, 2000].

En la práctica, se puede observar que la calidad media y la calidad mínima tienden a estabilizarse a medida que crece el número de etapas, por lo que transcurrido el suficiente número de iteraciones el proceso podría pararse automáticamente.

En el mismo artículo [Escobar et al., 2005] también se propone combinar la optimización de mallas comentada con el refinamiento local de este trabajo. La idea consiste en incrementar el número de nodos, y por lo tanto, los grados de libertad, en las regiones vecinas de los elementos con peor calidad. Se van a refinar todos los elementos cuya calidad es inferior a un cierto umbral y, una vez hecho esto, se iniciará una etapa de optimización hasta que la calidad de la malla alcance un cierto límite. El proceso completo se puede repetir varias veces hasta que se consiga la calidad requerida o no se mejore en absoluto.

Se han obtenido resultados por [Pain et al., 2001] y [Tam et al., 2000] usando técnicas de refinamientos/desrefinamientos en 3-D de mallas encajadas con el movimiento de nodos, en función de la forma y el tamaño de elementos en problemas anisotrópicos.

5.2.2. Optimización de mallas con funciones objetivo mejoradas

Las técnicas que habitualmente se emplean para mejorar una malla *válida* (sin elementos invertidos) están basadas en un suavizado local. Consiste en encontrar unas nuevas posiciones para los nodos que optimicen una función objetivo. Esa función está basada en una cierta medida de calidad [Knupp, 2001] de la *submalla local* $N(v)$, formada por el conjunto de tetraedros conectados al *nodo libre* v. Puesto que es un proceso local, no se puede garantizar que se alcance un óptimo global para toda la malla. Sin embargo, repitiendo este proceso varias veces para todos los nodos de la malla se pueden conseguir resultados satisfactorios.

Esas funciones objetivo [Knupp, 2000] son apropiadas para mejorar la calidad de un malla válida, pero no son adecuadas cuando hay elementos invertidos. Esto sucede porque presentan singularidades (barreras) cuando algún tetraedro de $N(v)$ cambia el signo de su jacobiano. La barrera evita que puedan aparecer elementos invertidos en la optimización de la submalla pero, por otro lado, impide que la función trabaje adecuadamente cuando la malla está enredada. Si el nodo libre está fuera de la *región factible* (subconjunto de $\mathbb{R}^3$ donde puede estar situado v para que $N(v)$ sea una malla válida) la barrera impide alcanzar el mínimo necesario. Incluso puede suceder que la región factible no exista, por ejemplo, cuando los límites de la submalla están enredados. En esas situaciones estas funciones objetivo no están bien definidas en todo $\mathbb{R}^3$, por lo que no son

adecuadas para mejorar la calidad de la malla.

Para solventar este problema se puede actuar como [Freitag y Knupp, 2002; Freitag y Plassmann, 2000], donde el método de optimización consiste en dos etapas. En la primera, los elementos invertidos son desenredados por un algoritmo que maximiza su jacobiano negativo [Freitag y Plassmann, 2000]; en la segunda, la malla resultante es suavizada según otra función objetivo basada en una medida de calidad de los tetraedros de $N(v)$ [Freitag y Knupp, 2002]. Después del procedimiento de desenredo, la malla tiene una calidad muy mala porque la técnica no tiende a generar elementos de buena calidad y, tal como se indica en [Freitag y Knupp, 2002], no se pueden emplear algoritmos basados en el gradiente para para optimizar la función objetivo, ya que ésta no es continua en todo $\mathbb{R}^3$, haciendo que sea necesario emplea métodos no estándar.

En la sección 5.2.2.2 se muestra una alternativa a estos métodos, de forma que el desenredo y suavizado se realizan en la misma etapa, empleando una función objetivo modificada que es regular en todo $\mathbb{R}^3$. Además, cuando existe la región factible, el mínimo de la función original y de la modificada son casi idénticos, y cuando no existe ésta, el mínimo de la función modificada se sitúa en una posición que tiende a desenredar $N(v)$. Esto último ocurrire cuando los límites de $N(v)$ están enredados. Haciendo uso de esta modificación, se puede emplear un método de optimización estándar y sin restricciones para encontrar el mínimo de la nueva función objetivo [Bazaraa et al., 1993].

5.2.2.1. Funciones objetivo

Las funciones objetivo se pueden construir partiendo de alguna medida de calidad de los tetraedros [Dompierre et al., 1998]. Sin embargo, aquellas que se obtienen mediante operaciones algebraicas son especialmente adecuadas para el propósito comentado ya que el coste computacional requerido para su evaluación puede ser muy bajo.

Sea T un elemento tetraédrico en el espacio físico cuyos vértices están dados por $\mathbf{x}_k = (x_k, y_k, z_k)^T \in \mathbb{R}^3$, $k = 0, 1, 2, 3$ y sea T_R el elemento de referencia con vértices en $\mathbf{u}_0 = (0,0,0)^T$, $\mathbf{u}_1 = (1,0,0)^T$, $\mathbf{u}_2 = (0,1,0)^T$ and $\mathbf{u}_3 = (0,0,1)^T$. Si elegimos $\mathbf{x}_0$ como vector de traslación, la aplicación afín que transforma T_R en T es $\mathbf{x} = A\mathbf{u} + \mathbf{x}_0$, donde A es la matriz jacobiana de la aplicación referida al nodo $\mathbf{x}_0$, y expresada como $A = (\mathbf{x}_1 - \mathbf{x}_0, \mathbf{x}_2 - \mathbf{x}_0, \mathbf{x}_3 - \mathbf{x}_0)$.

Sea ahora T_I un tetraedro equilátero de lado unitario y cuyos vértices es-

tán situados en $\mathbf{v}_0 = (0,0,0)^T$, $\mathbf{v}_1 = (1,0,0)^T$, $\mathbf{v}_2 = (1/2, \sqrt{3}/2, 0)^T$, $\mathbf{v}_3 = \left(1/2, \sqrt{3}/6, \sqrt{2}/\sqrt{3}\right)^T$. Sea $\mathbf{v} = W\mathbf{u}$ la aplicación lineal que transforma T_R en T_I, siendo $W = (\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ su matriz jacobiana.

La aplicación afín que transforma T_I en T está dada por

$$\mathbf{x} = AW^{-1}\mathbf{v} + \mathbf{x}_0$$

y su matriz jacobiana es $S = AW^{-1}$. La matriz S es independiente del nodo elegido como referencia; se puede decir que es *invariante nodal* [Knupp, 2001]. Se pueden emplear la norma, determinante o la traza de S para construir medidas algebraicas de calidad de T. Por ejemplo, la norma de Frobenius de S, definida por $|S| = \sqrt{\mathrm{tr}\,(S^T S)}$, se puede calcular fácilmente, por lo que resulta muy adecuada para la optimización de mallas. En [Knupp, 2001] se muestra que $q = \frac{3\sigma^{\frac{2}{3}}}{|S|^2}$ es una medida algebraica de calidad de T, donde $\sigma = \det(S)$. El máximo valor que puede tomar esta medida es la unidad y corresponde al tetraedro equilátero. Además, cualquier tetraedro plano o degenerado tiene medida nula. Esta medida de calidad permite obtener una función de optimización. Así, sea $\mathbf{x} = (x, y, z)^T$ la posición del nodo libre v, y S_m la matriz jacobiana correspondiente al m-ésimo tetraedro de $N(v)$. Definiremos la función objetivo de $\mathbf{x}$ asociada al m-ésimo tetraedro como

$$\eta_m = \frac{|S_m|^2}{3\sigma_m^{\frac{2}{3}}} \tag{5.1}$$

La función objetivo correspondiente a $N(v)$ se puede construir a partir de la p-norma de $(\eta_1, \eta_2, \ldots, \eta_M)$ como

$$|K_\eta|_p(\mathbf{x}) = \left[\sum_{m=1}^{M} \eta_m^p(\mathbf{x})\right]^{\frac{1}{p}} \tag{5.2}$$

donde M es el número de tetraedros en $N(v)$. La función objetivo $|K_\eta|_1$ fue empleada por [Bank y Smith, 1997] para suavizar y adaptar mallas en 2-D. La misma función fue empleada por [Djidjev, 2000] para el suavizado tanto en 2-D como en 3-D. Esta función, junto con otras, se estudia y compara en [Knupp, 2000]. Se debe destacar que en las referencias citadas se emplea esta función objetivo sólo en mallas válidas.

Aunque esta función de optimización se comporta de manera suave en los puntos donde $N(v)$ es una submalla válida, presenta discontinuidades cuando

el volumen de algún tetraedro de $N(v)$ se acerca a cero. Esto es debido a que η_m tiende a infinito cuando σ_m tiende a cero y su numerador está acotado inferiormente. De hecho, se puede demostrar que $|S_m|$ alcanza un mínimo estrictamente positivo cuando v se coloca en el centro geométrico de la cara fija del tetraedro m-ésimo. La región en la que v puede estar situado para obtener una $N(v)$ válida (región factible), es el interior del conjunto poliédrico P, definido como $P = \bigcap_{m=1}^{M} H_m$, donde H_m son los semiespacios determinados por $\sigma_m(\mathbf{x}) \geqslant 0$. Este conjunto puede ser vacío, por ejemplo, cuando los límites de $N(v)$ están enredados. Es estos caso, la función $|K_\eta|_p$ deja de ser aplicable. Por otro lado, cuando la región factible, $int\ P \neq \emptyset$ existe, la función objetivo tiende a infinito cuando v se aproxima a los límites de P. Debido a esta singularidad, se forma una barrera que evita que los algoritmos basados en el gradiente de la función objetivo alcancen el mínimo si el nodo libre se encuentra fuera de la región factible. En otras palabras, con estas funciones objetivo no se puede optimizar una malla $N(v)$ enredada.

5.2.2.2. Funciones objetivo modificada

Se propone una modificación de la función anterior (5.2), de forma que la barrera asociada con la singularidad se elimine y la nueva función sea suave en todo $\mathbb{R}^3$. Un requisito esencial es que el mínimo de la función original y de la modificada sean casi idénticos cuando $int\ P \neq \emptyset$. La proposición consiste en sustituir σ en (5.2) por la función positiva y creciente

$$h(\sigma) = \frac{1}{2}(\sigma + \sqrt{\sigma^2 + 4\delta^2}) \tag{5.3}$$

siendo el parámetro $\delta = h(0)$. La función $h(\sigma)$ se puede ver representada en la figura 5.4. Con la modificación anterior, la nueva función objetivo vendrá dada por

$$\left|K_\eta^*\right|_p(\mathbf{x}) = \left[\sum_{m=1}^{M} (\eta_m^*)^p(\mathbf{x})\right]^{\frac{1}{p}} \tag{5.4}$$

donde

$$\eta_m^* = \frac{|S_m|^2}{3h^{\frac{2}{3}}(\sigma_m)} \tag{5.5}$$

es la función objetivo modificada para el tetraedro m-ésimo.

El comportamiento de $h(\sigma)$ en función del parámetro δ es tal que, $\lim_{\delta\to 0} h(\sigma) = \sigma, \forall\sigma \geq 0$ y $\lim_{\delta\to 0} h(\sigma) = 0, \forall\sigma \leq 0$. Así, si $int\ P \neq \emptyset$, entonces $\forall \mathbf{x} \in int\ P$ tenemos $\sigma_m(\mathbf{x}) > 0$, para $m = 1, 2, \ldots, M$ y, a medida que vayamos eligiendo valores de δ más pequeños, $h(\sigma_m)$ se va pareciendo más a σ_m, de manera que la función original y su correspondiente versión modificada son muy próximas en la región factible. Así, en dicha región, $\left|K_\eta^*\right|_p$ converge puntualmente a $|K_\eta|_p$ cuando $\delta \to 0$. Además, considerando que $\forall\sigma > 0$, $\lim_{\delta\to 0} h'(\sigma) = 1$ y $\lim_{\delta\to 0} h^{(n)}(\sigma) = 0$, para $n \geq 2$, es fácil comprobar que las derivadas de la función objetivo verifican la misma propiedad de convergencia. Como resultado de estas consideraciones, se concluye que las posiciones de v que minimizan la función objetivo original y la modificada son casi idénticas cuando el valor de δ es *pequeño*. En realidad, δ se selecciona en función del punto v bajo consideración, haciéndolo tan pequeño como sea posible pero de manera que la evaluación del mínimo de la función objetivo modificada no presente ningún problema computacional.

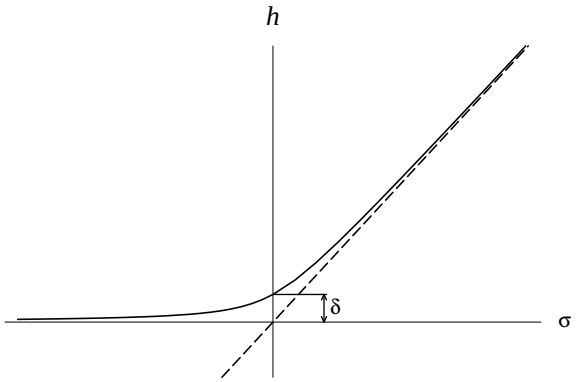


Figura 5.4: Representación de la función $h(\sigma)$

Si se supone que $int\ P = \emptyset$, entonces la función objetivo original, $|K_\eta|_p$, no es adecuada para el propósito buscado ya que no está correctamente definida. Sin embargo, la función objetivo modificada está correctamente definida y tiende a resolver el enredo. Podemos razonar esto desde un punto de vista cualitativo considerando que los términos dominantes en $\left|K_\eta^*\right|_p$ son aquellos que están asociados a tetraedros con valores de σ más negativos y, por ello, la minimización de estos términos implica el incremento de estos valores. Debemos remarcar que $h(\sigma)$ es una función creciente y $\left|K_\eta^*\right|_p$ tiende a ∞ cuando el volumen de cualquier tetraedro de $N(v)$ tiende a $-\infty$, ya que $\lim_{\sigma\to-\infty} h(\sigma) = 0$.

En resumen, mediante la función objetivo modificada es posible desenredar la malla a la vez que se mejora su calidad. En [Escobar et al., 2003] se pueden encontrar más detalle de este procedimiento de optimización.

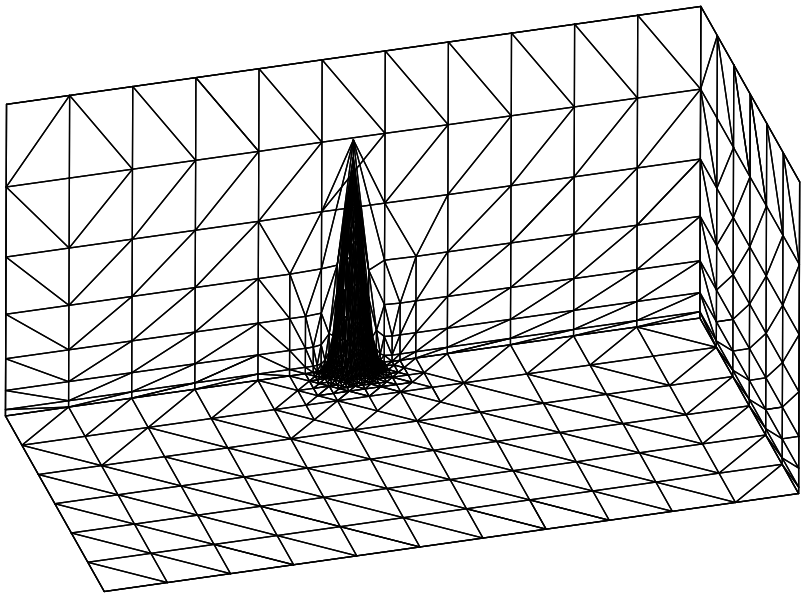
5.2.3. Experimentos numéricos

Con el fin de mostrar la efectividad de la combinación refinamiento/suavizado, se usará el siguiente problema test. Se partirá de una malla inicial M_0 con 1364 nodos y 5387 tetraedros. La malla se ha generado a partir de [Montenegro et al., 2002b,a] y contiene 43 tetraedros invertidos. El generador de mallas está basado en técnicas en 2-D de refinamiento/desrefinamiento [Ferragut et al., 1994] y una versión de la triangulación de Delaunay en 3-D [Escobar y Montenegro, 1996]. En la figura 5.5(a) se muestra un detalle de la malla con elementos invertidos y de baja calidad.

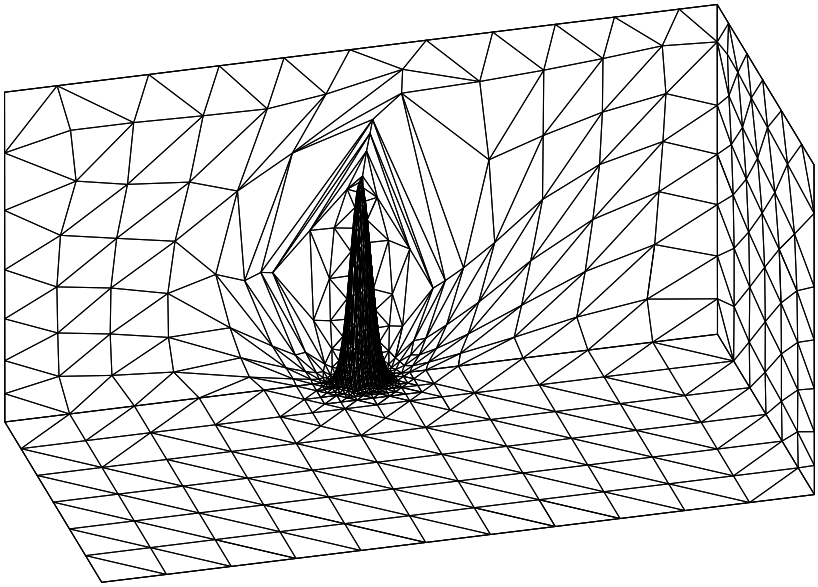
La figura 5.5(b) representa la malla desenredada y suavizada M_0', resultado de aplicar un cierto número de pasos del proceso de optimización hasta que los valores de la calidad mínima (q_{min}) y de las calidad media (q_{med}) permanecen estables en $q_{med} = 0.6714$ y $q_{min} = 0.0925$, respectivamente. En este proceso de optimización no se ha permitido que ningún nodo sea movido sobre la frontera inferior del dominio. En M_0' se pueden ver elementos con baja calidad en la vecindad de la zona del pico. Hay que destacar que la calidad de dichos elementos no se mejoraría manteniendo la misma topología de M_0 durante el proceso de optimización.

El procedimiento continúa refinando en M_0' los elementos con calidad cercana a $q_{min} = 0.0925$ mediante la división en 8-subtetraedros, y los adyacentes por conformidad. Tras este paso se obtendrá la malla M_1 (figura 5.6(a)), con 1438 nodos y 5758 tetraedros. Inicialmente, la calidad de la nueva malla es menor que la de la anterior. De hecho, los valores obtenidos para M_1 son $q_{med} = 0.6432$ y $q_{min} = 0.0702$.

Sin embargo, al aumentar en número de nodos en la vecindad de los elementos con peor calidad en M_0', se podría mejorar la calidad mínima realizando un suavizado sobre la malla refinada M_1. Tras este proceso se obtiene una malla suavizada M_1' con $q_{med} = 0.6499$ y $q_{min} = 0.1106$. Como puede verse, el valor de q_{min} aumenta respecto al de M_0', pero disminuye el de q_{med}. Realmente, en la mayoría de los casos es más adecuado aumentar la calidad mínima, o sea, mejorar la calidad de los elementos distorsionados. Además, el incremento re-



(a) M_0*: malla inicial enredada con* 43 *elementos invertidos*

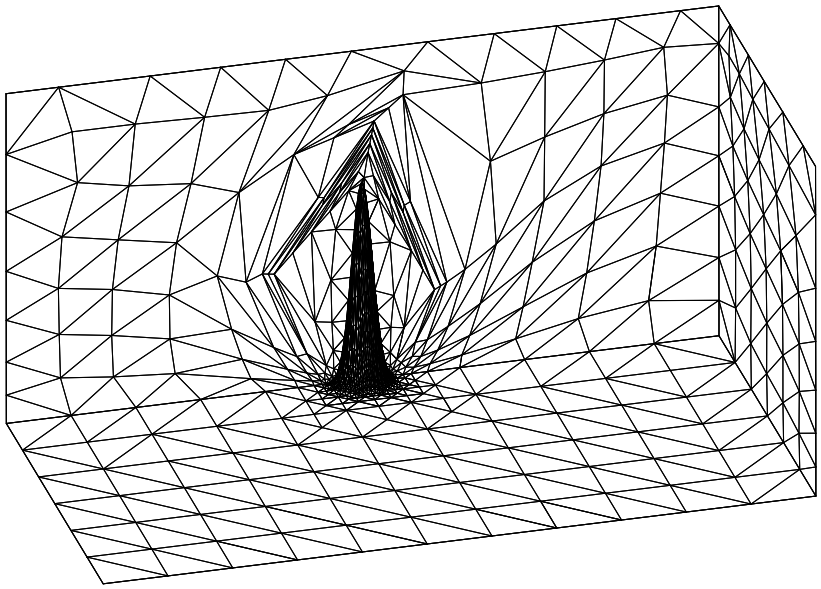


(b) M_0'*: malla desenredada y suavizada después de optimizar*

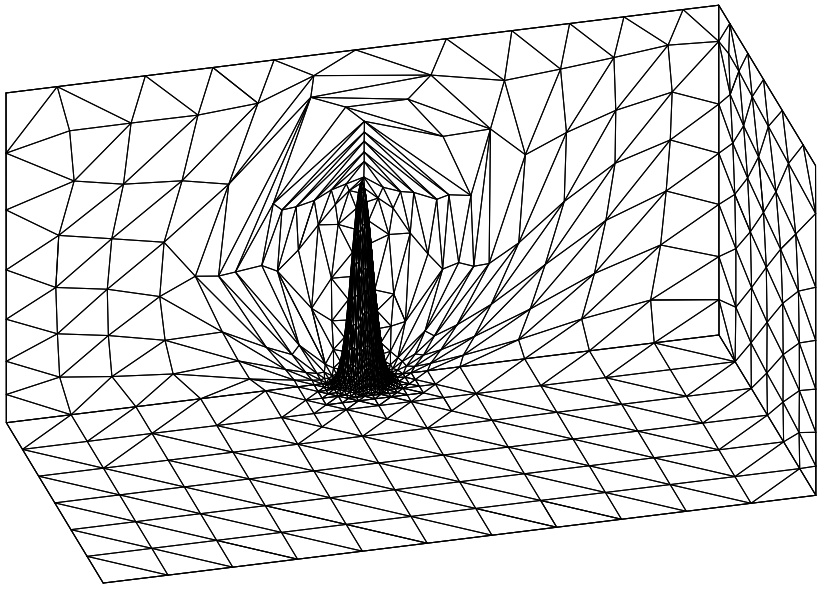
Figura 5.5: Malla inicial M_0 y malla desenredada y suavizada M_0'

lativo de q_{min} es mayor que la disminución relativa de q_{med}. En la figura 5.6(b) se muestra la malla M_1' donde puede apreciarse una mejora de la calidad en los elementos cercano a la superficie del pico.

Repitiendo el proceso de refinamiento a partir de M_1', se obtendrá la malla M_2 con 1475 nodos, 5925 tetraedros, $q_{med} = 0.6396$ y $q_{min} = 0.0924$. Suavizando



(a) M_1*: malla obtenida tras refinar* M_0'

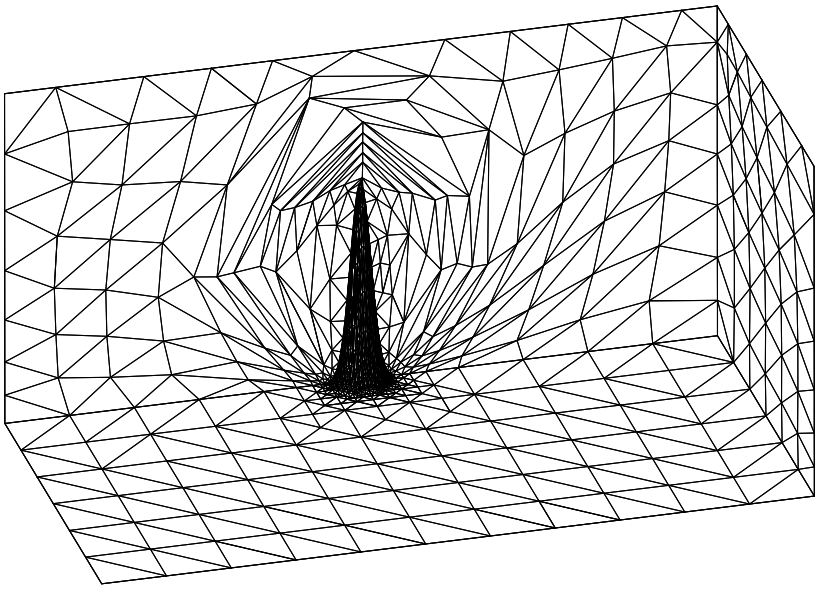


(b) M_1'*: malla* M_1 *suavizada*

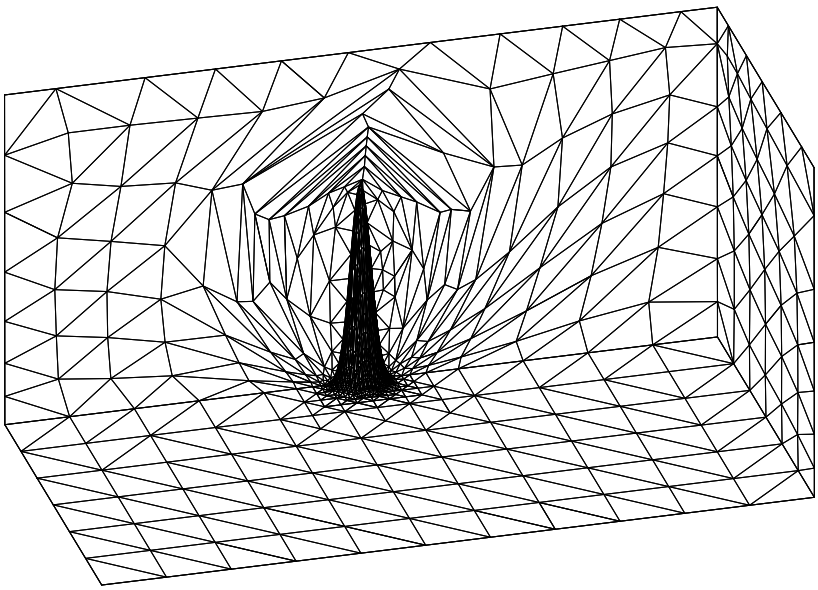
Figura 5.6: Resultados de aplicar refinamiento y suavizado sobre M_0'

esta malla se llegará a M_2' with $q_{med} = 0.6464$ and $q_{min} = 0.1214$.

Este último resultado implica que desde M_0 a M_2' se ha aumentado la calidad mínima en un 31.2 % con la introducción de unos pocos nodos. Por otro lado, la calidad media sólo ha empeorado en un 3.7 %. Las mallas M_2 y M_2' se pueden ver en la figuras 5.7(a) y 5.7(b), respectivamente.



(a) M_2*: malla obtenida tras refinar* M_1'



(b) M_2'*: malla* M_2 *suavizada*

Figura 5.7: Resultados de aplicar refinamiento y suavizado sobre M_1'

En resumen:

1. Se ha usado una estrategia concreta a la hora de asegurar la conformidad de las mallas refinadas. Si cualquier tetraedro transitorio que deba ser generado por razones de conformidad tuviera una calidad inferior al límite establecido para dividir en 8, se eliminaría la división interna de su

tetraedro padre y se marcaría éste como *Tipo I*. Además, cada vez que se aplica el algoritmo de refinamiento sobre una malla (refinada previamente) se considerará que no hay transitorios, es decir, se tratará como una malla inicial.

2. Se ha aplicado el procedimiento de refinamiento/suavizado una vez que la malla ha sido desenredada. Un posible caso se estudio es analizar qué ocurriría si se aplicara este método directamente sobre mallas enredadas.

3. En el problema test no se ha permitido movimiento de nodos sobre el límite inferior del dominio. Si se suprimiera esta restricción, la calidad del resultado mejoraría, tal y como se puede comprobar en [Escobar et al., 2006].

5.2.4. Conclusiones

La combinación de técnicas de suavizado con algoritmos de refinamiento local se ha mostrado útil para mejorar la calida mínima de los elementos de mallas de tetraedros con muy baja calidad. Además, como la estrategia planteada refina pocos elementos en cada etapa, el número de nuevos nodos añadidos en la malla inicial es muy bajo respecto de número total. Por supuesto, se podría repetir la combinación de refinamiento/suavizado hasta que se obtenga la calidad deseada o no se consiga mejorar. El límite de la calidad vendrá dado por la topología de la malla inicial y por las restricciones impuestas en las fronteras del dominio.

5.3. Modelización de campos de viento

Los modelos para ajustes de campos de viento se suelen aplicar sobre orografía complejas. Parten de un discretización tridimensional de la zona de estudio, y el resultado final será un campo de velocidades para todo el dominio. Los algoritmos de refinamiento/desrefinamiento se emplean para mejorar la solución en zonas de la malla en la que se observe una fuerte variación de la solución, o para suprimir elementos cuando la solución numérica tenga poca diferencia entre elementos adyacentes.

En [Montero et al., 2005] se muestra un modelo de viento en el que aplican algoritmos genéticos y refinamiento para ajustar los parámetros del mismo.

Partiendo de una malla de tetraedros con una gran densidad de nodos cerca del terreno, generada con [Montenegro et al., 2002b], se presenta un modelo de masa consistente que genera un campo de velocidades para un fluido incompresible con condiciones de impenetrabilidad en la superficie del terreno, a partir de un campo de velocidad inicial obtenido de medidas experimentales y ciertas consideraciones físicas. El primer paso será construir el campo de velocidad inicial, realizando una interpolación horizontal a la altura de las estaciones de medida sobre el terreno. Con esos datos, se elabora un perfil vertical teniendo en cuenta la estabilidad atmosférica, la rugosidad del terreno, el viento geostrófico y la estratisficación atmosférica. A continuación se formularán las ecuaciones de continuidad de masa para un fluido incompresible mediante funciones de mínimos cuadrados. Los multiplicadores de Lagrange llevan a un problema elíptico que se resolverá aplicando el método de los elementos finitos. A partir de aquí se podría aplicar técnicas de refinamiento para mejorar la solución numérica. Una explicación detallada del modelo de viento se puede encontrar en [Rodríguez Barrera, 2004].

Lo que se pretende obtener en [Montero et al., 2005], en definitiva, son los valores numéricos de los parámetros asociados al modelo de viento. Para este fin, se han empleado algoritmos genéticos.

5.3.1. Modelo de masa consistente

Este modelo [Rodríguez et al., 2002] está basado en la ecuación de continuidad para un fluido incompresible donde la densidad del aire es constante en el dominio Ω, una campo de velocidades $\vec{u}$ y considerando condiciones de impenetrabilidad en Γ_b (terreno y límite superior del dominio):

$$\vec{\nabla} \cdot \vec{u} = 0 \qquad \text{en } \Omega \tag{5.6}$$

$$\vec{n} \cdot \vec{u} = 0 \qquad \text{en } \Gamma_b \tag{5.7}$$

Se formula un problema de mínimos cuadrado en Ω con las velocidades a ajustar $\vec{u}(\widetilde{u}, \widetilde{v}, \widetilde{w})$

$$E(\vec{u}) = \int_\Omega \left[\alpha_1^2 \left((\widetilde{u} - u_0)^2 + (\widetilde{v} - v_0)^2\right) + \alpha_2^2 \left(\widetilde{w} - w_0\right)^2\right] \, d\Omega \tag{5.8}$$

siendo $\vec{v}_0 = (u_0, v_0, w_0)$ el viento interpolado obtenido de medidas experimenta-

les y α_1, α_2 son los módulos de precisión de Gauss. Este problema es equivalente a encontrar el punto silla $(\vec{v}, \phi)$ del Lagrangiano [Winter et al., 1995]

$$E(\vec{v}) = \min_{\vec{u} \in K} \left[E(\vec{u}) + \int_\Omega \phi \vec{\nabla} \cdot \vec{u} \, d\Omega \right] \tag{5.9}$$

siendo $\vec{v} = (u, v, w)$, ϕ el multiplicador de Lagrange y K el conjunto de funciones admisibles. La técnica de los multiplicadores de Lagrange (5.9) se usa para minimizar (5.8) con las restricciones (5.6) y (5.7), obteniéndose

$$u = u_0 + T_h \frac{\partial \phi}{\partial x}, \quad v = v_0 + T_h \frac{\partial \phi}{\partial y}, \quad w = w_0 + T_v \frac{\partial \phi}{\partial z} \tag{5.10}$$

donde $T = (T_h, T_h, T_v)$ es el tensor diagonal de transmisión, con $T_h = \frac{1}{2\alpha_1^2}$ y $T_v = \frac{1}{2\alpha_2^2}$. Puesto que α_1 y α_2 son constantes en Ω, sustituyendo (5.10) en (5.6), resulta el siguiente problema elíptico para el multiplicador de Lagrange

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{T_v}{T_h} \frac{\partial^2 \phi}{\partial z^2} = -\frac{1}{T_h} \left(\frac{\partial u_0}{\partial x} + \frac{\partial v_0}{\partial y} + \frac{\partial w_0}{\partial z} \right) \quad \text{en} \quad \Omega \tag{5.11}$$

Se consideran condiciones tipo Dirichlet nula (5.12) para las fronteras permeables (verticales del dominio) y tipo Neumann (5.13) para terreno y límite superior

$$\phi = 0 \quad \text{en} \quad \Gamma_a \tag{5.12}$$

$$\vec{n} \cdot T \vec{\nabla} \mu = -\vec{n} \cdot \vec{v}_0 \quad \text{en} \quad \Gamma_b \tag{5.13}$$

Para la resolución del problema dado por (5.11), (5.12) y(5.13) se utiliza una malla de tetraedros [Montenegro et al., 2002b], lo que lleva a un conjunto de matrices elementales de dimensión 4×4 y de vectores elementales de 4×1 . Se ensamblan para formar un sistema de ecuaciones simétrico lineal que se resuelve por el método del gradiente conjugado con precondicionamiento [Rodríguez Barrera, 2004].

5.3.2. Viento interpolado

El primer paso para construir el viento $\vec{v}_0$ es la interpolación horizontal a partir de medidas de velocidad de viento a la altura de las estaciones z_m usando

la distancia horizontal y la diferencia de altura entre el punto considerado y las estaciones [Montero et al., 1998]

$$\vec{v}_0(z_m) = \varepsilon \frac{\sum_{n=1}^{N} \frac{\vec{v}_n}{d_n^2}}{\sum_{n=1}^{N} \frac{1}{d_n^2}} + (1-\varepsilon) \frac{\sum_{n=1}^{N} \frac{\vec{v}_n}{|\Delta h_n|}}{\sum_{n=1}^{N} \frac{1}{|\Delta h_n|}} \tag{5.14}$$

donde $\vec{v}_n$ es la velocidad observada en la estación n, N es el número de estaciones consideradas en la interpolación, d_n es la distancia horizontal desde la estación n al punto en el que se quiere calcular la velocidad, $|\Delta h_n|$ es la diferencia de altura entre la estación n y el punto de estudio y ε es un parámetro de peso $(0 \leq \varepsilon \leq 1)$, que permite dar mayor importancia a la interpolación inversa al cuadrado de la distancia o inversa a la diferencia de altura.

En el perfil vertical del viento se asume que este modelo no tiene en cuenta los fenómenos de turbulencias cerca del terreno debido a su rugosidad. Por lo tanto, se establece velocidad nula por debajo de la longitud de rugosidad z_0

$$\vec{v}_0(z) = 0 \qquad z \leq z_0 \tag{5.15}$$

Se ha considerado un perfil logarítmico en la capa superficial que tiene en cuenta la interpolación horizontal anterior, así como el efecto de la rugosidad y la estabilidad atmosférica (neutra, estable o inestable, según la clasificación de estabilidad de Pasquill) en la dirección e intensidad del viento. Sobre la capa superficial, se realiza una interpolación lineal usando el viento geostrófico. El perfil logarítmico viene dado por

$$\vec{v}_0(z) = \frac{\vec{v}^*}{k} \left(\log \frac{z}{z_0} - \Phi_m \right) \qquad z_0 < z \leq z_{sl} \tag{5.16}$$

donde $\vec{v}^*$ es la velocidad de fricción, k es la constante de von Karman y z_{sl} es la altura de la capa superficial. El valor de Φ_m depende de la estabilidad del aire

$$\begin{aligned} \Phi_m &= 0 && \text{(neutra)} \\ \Phi_m &= -5\frac{z}{L} && \text{(estable)} \\ \Phi_m &= \log\left[\left(\frac{\theta^2+1}{2}\right)\left(\frac{\theta+1}{2}\right)^2\right] - 2\arctan\theta + \frac{\pi}{2} && \text{(inestable)} \end{aligned} \tag{5.17}$$

donde $\theta = (1-16\frac{z}{L})^{1/4}$ y $\frac{1}{L} = az_0^b$, con a y b dependiendo de la clasificación de

estabilidad de Pasquill. L es la longitud de Monin-Obukhov. La velocidad de fricción se obtiene en cada punto de las medidas interpoladas a la altura de las estaciones (interpolación horizontal)

$$\vec{v}^* = \frac{k\ \vec{v}_0(z_m)}{\log \frac{z_m}{z_0} - \Phi_m} \tag{5.18}$$

La altura de la capa límite planetaria z_{pbl} sobre el terreno se escoge de forma que la intensidad y dirección de viento sean constantes en esa altura

$$z_{pbl} = \frac{\gamma\ |\vec{v}^*|}{f} \tag{5.19}$$

donde $f = 2\omega \sin\varphi$ es el parámetro de Coriolis (ω es la rotación de la tierra y φ la latitud), y γ es un parámetro dependiente de la estabilidad atmosférica. La altura de mezcla h coincide con z_{pbl} en condiciones neutras e inestables. En condiciones estables, Zilitinkevich sugiere ([Businger y Arya, 1974])

$$h = \gamma' \sqrt{\frac{|\vec{v}^*|\ L}{f}} \tag{5.20}$$

donde γ' es otra constante de proporcionalidad. La altura de la capa superficial es $z_{sl} = \frac{h}{10}$. Desde z_{sl} a z_{pbl}, se realiza una interpolación lineal con viento geostrófico $\vec{v}_g$

$$\vec{v}_0(z) = \rho(z)\,\vec{v}_0(z_{sl}) + [1-\rho(z)]\vec{v}_g \qquad z_{sl} < z \le z_{pbl} \tag{5.21}$$

$$\rho(z) = 1 - \left(\frac{z - z_{sl}}{z_{pbl} - z_{sl}}\right)^2 \left(3 - 2\frac{z - z_{sl}}{z_{pbl} - z_{sl}}\right) \tag{5.22}$$

Finalmente, en el modelo se asume que

$$\vec{v}_0(z) = \vec{v}_g \qquad z > z_{pbl} \tag{5.23}$$

5.3.3. Refinamiento adaptativo

Para determinar qué elementos de la malla deberán ser refinados se va emplear un indicador de error atendiendo al gradiente de la solución en cada elemento. Partiendo de una malla inicial obtenida con [Montenegro et al., 2002b; Escobar y Montenegro, 1996], se aplicará el refinamiento por división en 8-

subtetraedros aplicando el indicador de error mencionado.

En la malla T_j, se asociará un indicador de error η_i^j al elemento $\tau_i^j \in T_j$ tipo gradiente definido como

$$\eta_i^j = (d_i)^p \left| \vec{\nabla} \phi_h \right| \tag{5.24}$$

donde se asume que el parámetro p es generalmente 1 or 2, y d_i es la longitud de la arista mayor del tetraedro τ_i^j. Si $p = 1$ y se considera interpolación lineal en los elementos de T_j, entonces η_i^j representa un límite superior de la variación máxima de ϕ_h en el elemento τ_i^j. Una vez que η_i^j ha sido calculado, un elemento debe ser refinado si $\eta_i^j \geq \gamma \eta_{\text{máx}}^j$, siendo $\gamma \in [0, 1]$ el parámetro de refinamiento y $\eta_{\text{máx}}^j$ el valor máximo de los indicadores de todos los elementos de T_j.

5.3.4. Estimación de parámetros

Cuatro son los parámetros del modelo de viento que sería interesante estimar. En primer lugar, el parámetro de estabilidad:

$$\alpha = \frac{\alpha_1}{\alpha_2} = \sqrt{\frac{T_v}{T_h}} \tag{5.25}$$

puesto que el mínimo del funcional de (5.8) es el mismo si se divide por α_2^2. Por otro lado, para $\alpha >> 1$, predomina el ajuste del flujo en dirección vertical, mientras que para $\alpha << 1$, se potencia el ajuste horizontal. Por lo tanto, la elección de α permite que el aire tienda a sobrepasar las barreras del terreno o alrededor de ellas respectivamente [Ratto, 1996b]. Incluso el comportamiento del modelo de masa consistente en la mayoría de los experimentos numéricos se muestra muy sensible al valor escogido para α, por lo que se debe tener muy en cuenta. Muchos autores han estudiado la parametrización de la estabilidad ya que la dificultad de determinar valores de α ha limitado el empleo del modelo en orografías complejas. En [Sherman, 1978], [Kitada et al., 1983] y [Davis et al., 1984], se propone tomar $\alpha = 10^{-2}$, es decir, proporcional a la magnitud de w/u. Otros autores, [Ross et al., 1988] y [Moussiopoulos et al., 1988], vinculan α con el número de Froude, mientras que [Geai, 1985], [Lalas et al., 1988] y [Tombrou y Lalas, 1990] proponen que α varíe en dirección vertical. Finalmente, [Barnard et al., 1987] proponen un método para obtener α en cada etapa de simulación. La idea es usar N velocidades de viento observadas para obtener el campo de viento y usar las restantes, N_r, como referencia. Entonces se realizarán

diversas simulaciones con distintos valores de α. El valor que más acerque el viento estimado al observado en las estaciones de referencia es el que se toma como valor del parámetro de estabilidad. Este método proporciona valores de α que sólo son válidos para cada caso particular y, por tanto, no proporciona valores adecuados, a priori, para otras simulaciones. En [Montero et al., 2005] y [Rodríguez Barrera, 2004] se estudia una versión del método propuesto en [Barnard et al., 1987], utilizando algoritmos genéticos como herramienta de optimización que permite una selección automática de α.

El segundo parámetro que va a estimarse es el coeficiente de peso ε ($0 \leq \varepsilon \leq 1$) de la ecuación (5.14), correspondiente a la interpolación horizontal de las medidas de viento observadas. Cuando $\varepsilon \to 1$, adquiere más importancia la distancia horizontal de cada punto a las estaciones de medida, mientras que para $\varepsilon \to 0$ se da más peso a la distancia vertical entre cada punto y las estaciones [Montero et al., 1998]. En general, para terrenos complejos se utiliza la segunda aproximación [Palomino y Martín, 1995]. En orografías más llanas o en análisis horizontales en 2-D, se utiliza la primera. En aplicaciones más realistas existirán zonas con orografía compleja y zonas de orografía más regular, lo que sugiere el uso de valores intermedios de ε.

El siguiente parámetro objeto de estimación es γ, que aparece en la ecuación (5.19) y está relacionado con la capa límite planetaria en la estratificación atmosférica. Existen diferentes autores que proponen distintos rangos para este parámetro. [Panofsky y Dutton, 1984] proponen el intervalo $[0.15, 0.25]$. Sin embargo, en [Ratto, 1996a] se utiliza directamente el valor $\gamma = 0.3$ en el código de su programa *WINDS*, mientras que para [de Baas, 1996] ha de estar dentro del intervalo $[0.3, 0.4]$. En las simulaciones realizadas el espacio de búsqueda de γ incluye todas estas posibilidades.

Finalmente, también resulta de interés obtener estimaciones de los valores del parámetro γ', que interviene en el cálculo de la altura de la capa de mezcla en el caso de condiciones atmosféricas estables (5.20). [Garratt, 1982] propone directamente $\gamma' = 0.4$. También en el código de *WINDS* el valor de γ' está en torno a 0.4. Así, se define el espacio de búsqueda para el valor de γ' en el entorno de 0.4.

Los parámetros ε, γ y γ' intervienen en la interpolación del viento inicial, mientras que α afecta al cálculo del viento resultante.

5.3.5. Algoritmos genéticos

Los algoritmos genéticos (AG) son herramientas de optimización basadas en el mecanismo de evolución natural. Producen intentos sucesivos que tienen una probabilidad cada vez mayor de alcanzar el óptimo global. Los trabajos se basaron en el modelo de AG desarrollado por [Levine, 1994]. Los aspectos más importantes de los AG son la construcción de una población inicial, la evaluación de cada individuo a través de la función de aptitud, la selección de los padres de la siguiente generación, el cruce de esos padres para crear los hijos y la mutación, que incrementa la diversidad.

Normalmente se usan dos técnicas de reemplazamiento de la población. La primera, llamada reemplazo generacional, sustituye la población entera cada vez [Holland, 1992]. La segunda, que se conoce como de estado estacionario (*steadystate*), únicamente reemplaza unos pocos individuos en cada generación [Syswerda, 1989; Whitley, 1989, 1988]. Como criterios de parada se emplean los siguientes: número máximo de iteraciones, población demasiado homogénea y el hecho de que no cambie la mejor solución en un número dado de iteraciones. La población inicial normalmente se genera de forma aleatoria y la única propiedad que ha de verificar es que sus elementos sean lo suficientemente diversos.

La fase de selección crea una población intermedia a partir de la evaluación de la función de ajuste. Se seleccionaros dos esquemas [Levine, 1994]: Selección Universal Estocástica (SU) y Selección por Torneo Binario (BT). El operador de cruce elige bits de cada padre y los combina para crear un hijo. Se emplea el operador de Cruce Uniforme(U). Dependerá de la probabilidad de intercambio de dos bits de los padres [De Jong y Spears, 1992]. Esto permite obtener individuos en el espacio de búsqueda que de otra forma no serían evaluados. Cuando parte de un cromosoma se ha seleccionado para mutar, el gen correspondiente a esa parte se cambia también. Sucede con probabilidad p. Se trabajó con dos operadores de mutación. El primero con la forma $\nu \leftarrow \nu \pm p \times \nu$, donde ν es el valor del alelo existente, y p sale de una distribución Gaussiana (G). El segundo operador (R) simplemente reemplaza ν con un valor seleccionado uniformemente aleatorio del rango inicial de los genes.

La función de ajuste se hace relativa al medio ambiente. Evalúa cada individuo de la población. Esta es una medida relativa al resto de la población, acerca de lo bien que dicho individuo satisface el problema. Los valores se transforman a una escala de valores positivos crecientes monótonamente. En los experimen-

tos numéricos con el modelo de viento se buscó valores para los parámetros citados anteriormente. Para ello se pretende minimizar el error relativo medio de las velocidades calculadas por el modelo con respecto a las medidas en las estaciones

$$F(\alpha, \varepsilon, \gamma, \gamma') = \frac{1}{N_r} \sum_{n=1}^{N_r} \frac{|\vec{v}_n - \vec{v}(x_n, y_n, z_n)|}{|\vec{v}_n|} \tag{5.26}$$

donde $\vec{v}(x_n, y_n, z_n)$ es la velocidad del viento obtenida por el modelo en la posición de la estación n, y N_r es el número de estaciones de referencia.

5.3.6. Efecto de una chimenea en el campo de velocidades

La idea es incorporar al perfil inicial de velocidades de vientos, donde habitualmente sólo se calculan las componentes horizontales de las velocidades debido a la ausencia de medidas de la componente vertical en las estaciones, una componente vertical no nula en la trayectoria de una posible pluma de contaminantes originada por una fuente emisora (chimenea). De esta forma se pretende simular el campo de velocidades del fluido compuesto por dos aportaciones: la del viento y la de expulsión del gas contaminante de la chimenea. Los actuales modelos de pluma gaussiana permiten aproximar los valores de la altitud efectiva z_H de la pluma y la distancia horizontal d_f desde el centro de la superficie de salida de la chimenea hasta donde se alcanza z_H, en función de las características de la emisión, del viento y de la estabilidad atmosférica [Boubel et al., 1994].

Los gases que salen de una chimenea alcanzan una altura superior a la de la chimenea cuando estos son de menor densidad que el aire del entorno (elevación por flotación, figura 5.8) o bien son expulsados a una velocidad suficiente que les proporciona una energía cinética (elevación por momento, figura 5.9). La elevación por flotación se denomina en ocasiones elevación térmica ya que la causa más común de disminución de la densidad es el aumento de temperatura. Para estimar la altura efectiva de la pluma se utilizan las ecuaciones de [Briggs, 1969, 1971, 1972, 1973, 1975].

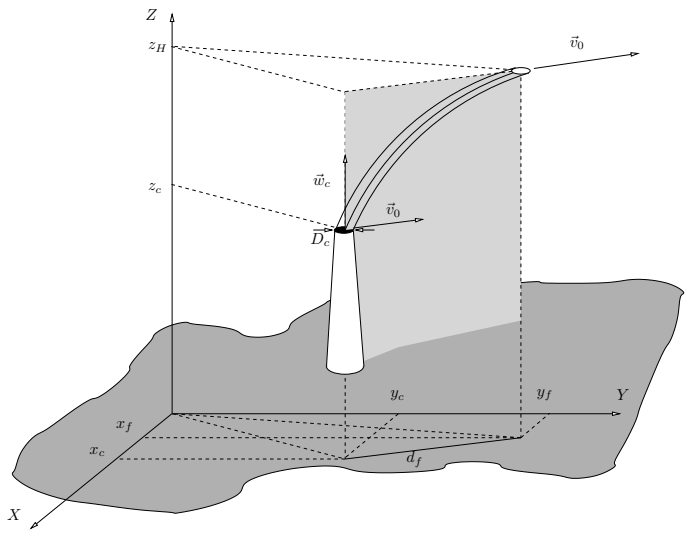


Figura 5.8: Zona de corrección de la componente vertical de la velocidad del fluido con elevación por flotación

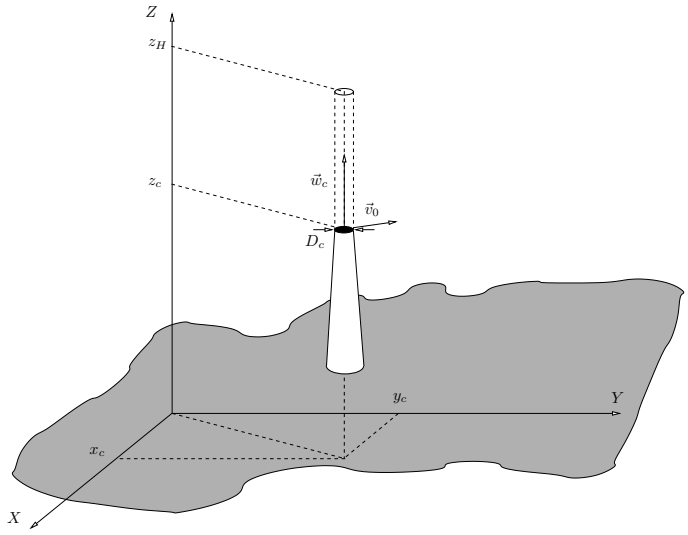


Figura 5.9: Zona de corrección de la componente vertical de la velocidad del fluido con elevación por momento

Se puede ver un desarrollo matemático detallado del modelo en [Rodríguez Barrera, 2004], [Montenegro et al., 2004] y [Montenegro et al., 2005].

5.3.7. Experimentos numéricos

Se han estudiado los mismos problemas (casos I y III) sobre la zona sur de la isla de La Palmas presentados en [Montero y Sanín, 2001; Rodríguez et al., 2002]. Un dominio real de $45600 \times 31200 \times 9000\ m^3$, discretizado con [Montenegro et al., 2002b]. La altura máxima en esa zona de la isla es de 2279 m. Se parte de una malla inicial M_0 con 11416 nodos y 55003 tetraedros. El refinamiento de dicha malla en las zonas de las estaciones de medida genera una nueva malla M_0' con 11494 nodos y 55363 tetraedros (figura 5.10). Este refinamiento se ha desarrollado atendiendo únicamente a consideraciones geométricas. Las medidas de viendo se han tomado en cuatro estaciones: MBI, MBII, MBIII y LPA. Para realizar los test en diferentes condiciones atmosféricas, en el caso I se ha considerado condiciones ligeramente inestables y en el caso III condiciones ligeramente estables. Puesto que el número de datos disponibles es bajo, se ha usado el viento observado en las estaciones MBI, MBII y LPA para calcular el campo de viento interpolado (5.14), es decir, $N = 3$, y las medidas de MBIII como referencias para la función de ajuste (5.26), es decir, $N_r = 1$.

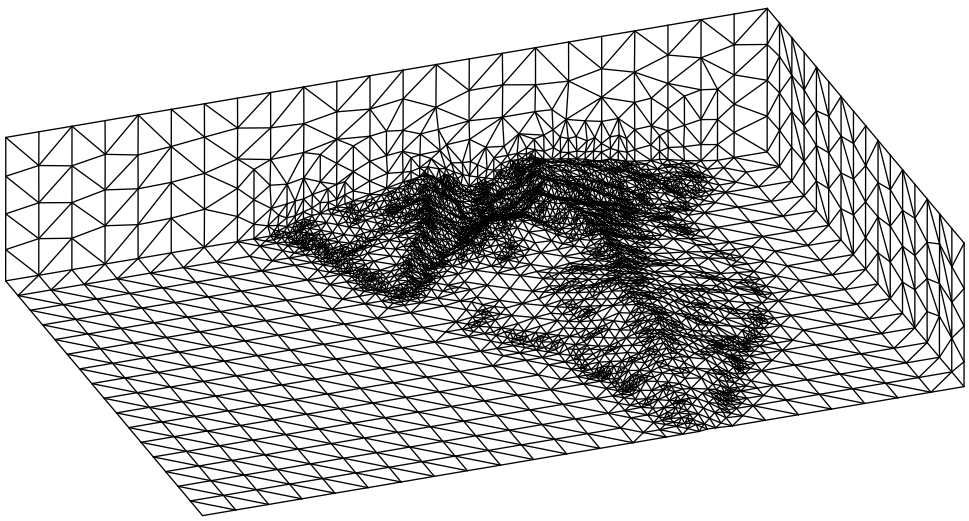


Figura 5.10: Detalle de la malla M_0'

En la primera aplicación (caso I), dadas las condiciones inestables de la atmósfera, el parámetro γ' no se usa en el modelo, ya que $h = z_{pbl}$. Por lo tanto, solo se necesita estimar α, ε y γ . Se ha dividido el experimento en dos etapas: primero se fija $\gamma = 0.3$ y se estima $\alpha \in [10^{-3}, 10]$ y $\varepsilon \in [0, 1]$.

La segunda columna de la tabla 5.1 (*Etapa 1*) muestra los valores obtenidos para α y ε, los cuales muestras un ajuste del viento casi vertical y la complejidad de la orografía respectivamente. Se debe destacar que se ha obtenido un error del modelo en la estación MBIII en torno al 4.96 %. Las estrategias de los AG (*BT, U, R*) se corresponden con los operadores de selección, cruce y mutación más eficientes de varios test con diferentes combinaciones. En la segunda etapa se estiman α, ε y $\gamma \in [0.15, 0.5]$. Los resultados de muestran en la tercera columne de la tabla 5.1. Se puede ver que α alcanza el máximo valor de su espacio de búsqueda, ε permanece en torno a 0.5 y γ disminuye, de forma que el error respecto a la estación MBIII es del 4.76 %. En este test la peor evaluación de la función de ajuste, correspondiente a valores de los parámetros dentro de su espacio de búsqueda, arroja un error del 68.07 % y 34.62 % respectivamente en cada etapa. Por lo tanto, conocer los valores adecuados de los parámetros es fundamental para que el modelo numérico sea eficiente.

	Etapa 1	Etapa 2
Estrategia de AG	BT, U, R	SU, U, G
Iteraciones	88	135
tiempo CPU (s)	10385	16194
Mejor ajuste	0.0496	0.0476
α	9.978	10.000
ε	0.609	0.484
γ	(0.300)	0.150

Tabla 5.1: Caso I: Estrategias de AG, mejor evaluación de la función y valor de parámetros (*valores fijos en paréntesis*)

Para el segundo experimento (caso III) se siguió un procedimiento similar. Es este caso se tendrá en cuenta $\gamma' \in [0.15, 0.5]$. En primer lugar, se resuelve un problema con dos parámetros (α, ε) desconocidos. En la segunda columna de la tabla 5.2 (*Estapa 1*) se pueden ver los valores obtenidos. A continuación, cuatro problemas en los que se fija uno de los parámetros cada vez (*Etapas 2-5*). Finalmente, se estiman los cuatro parámetros simultáneamente (*Etapa 6*). Las condiciones atmosféricas estables dejan el predominio del ajuste vertical que surge en el experimento anterior con condiciones inestables, así como aumenta la importancia de la distancia horizontal en la interpolación del viento observado. En la *Etapa 6*, el error mínimo obtenido en la estación MBIII es alrededor del 11.87 %, mientras que la peor evaluación fue del 994.2 %. En ambos experi-

mentos, el número de individuos de la población inicial fue de 100, excepto para la etapa 6 del caso III donde fue de 150.

	Etapa 1	Etapa 2	Etapa 3	Etapa 4	Etapa 5	Etapa 6
Estrategia de AG	SU, U, G	SU, U, R	SU, U, R	SU, U, R	SU, U, R	SU, U, R
Iteraciones	81	82	93	123	435	431
tiempo CPU (s)	9613	9478	10970	14758	50849	75692
Mejor ajuste	0.1810	0.1612	0.1248	0.1213	0.1191	0.1187
α	10.000	9.968	(9.968)	9.922	9.995	9.999
ε	0.672	0.780	0.808	(0.808)	0.810	0.808
γ	(0.300)	0.244	0.234	0.230	(0.230)	0.231
γ'	(0.400)	(0.400)	0.164	0.151	0.150	0.150

Tabla 5.2: Caso III: Estrategia AG, evaluación función y valor de parámetros (*valores fijos en paréntesis*)

En las tablas 5.1 y 5.2 se muestra el tiempo de CPU y el número de iteraciones en un cluster de 5 nodos Pentium 4 a 1.6 GHz, ejecutando dos procesos en cada nodo. Fue la mejor estrategia, la más rápida, aunque se intentó lanzar 3 y 4 procesos por nodo. Cabe destacar que la evaluación de un individuo de una generación implica la resolución de un problema de viento por elementos finitos usando dos pasos de refinamiento adaptativo.

En comparación con los resultados obtenidos en [Rodríguez et al., 2002], en los que no se usaban estrategias de refinamiento, se puede comprobar que el error se reduce a la mitad en cada test.

Como ejemplo, se considera una estrategia adaptativa para el cálculo del campo de viento en el segundo test usando los valores de los parámetro de la *Etapa 6*. En primer lugar, se refina M_0' usando en indicador de error dado por (5.24) con un parámetro de refinamiento de $\gamma = 0.4$. La malla resultante M_1 (figura 5.11) tiene 13135 nodos y 64684 tetraedros. Se repite la misma estrategia de refinamiento sobre M_1 para obtener M_2 con 19205 y 99422 tetraedros (figura 5.12). En este caso, las medidas de las cuatro estaciones se han tenido en cuenta para el viento interpolado. Las figuras 5.13 y 5.14 muestran el perfil dinámico y velocidad del viento obtenido por el modelo a 500 m de altura.

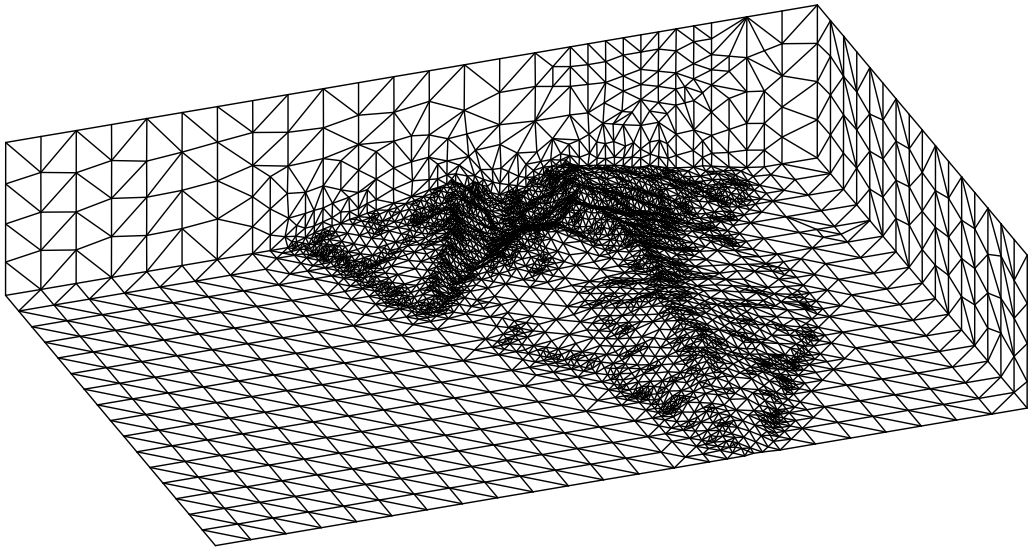


Figura 5.11: Malla refinada M_1

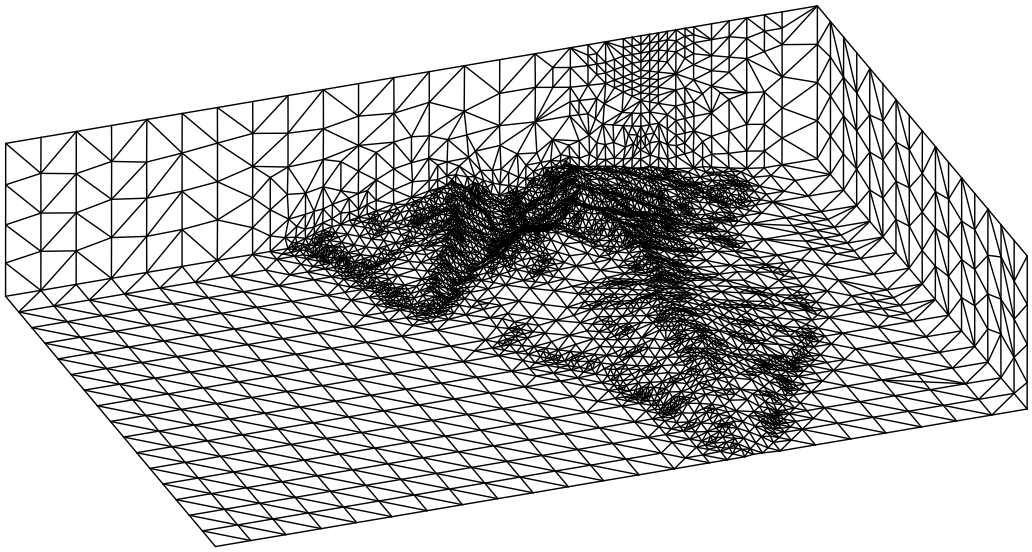


Figura 5.12: Malla refinada M_2

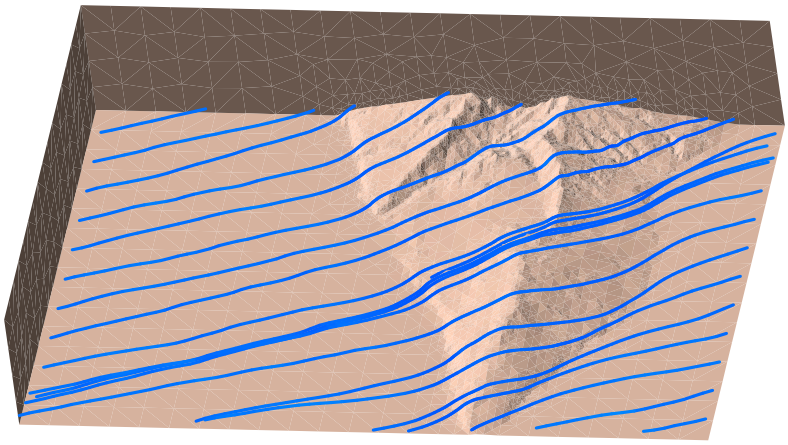


Figura 5.13: Perfil dinámico del viento del segundo test a 500 m de altura

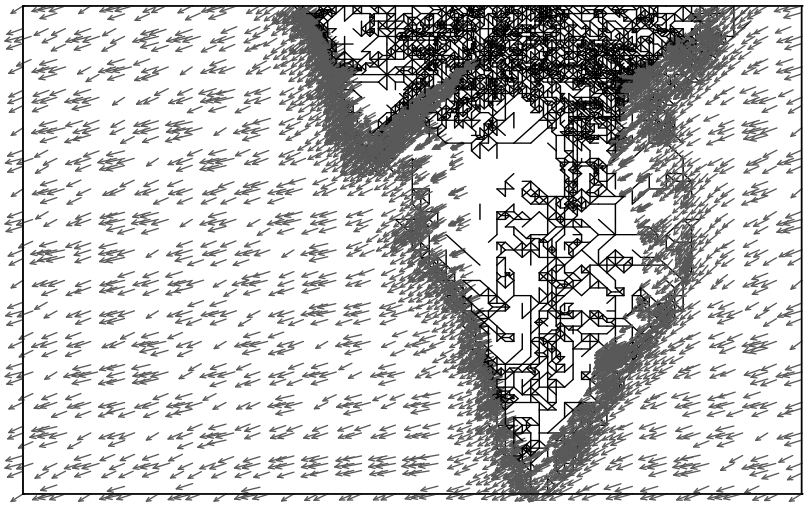


Figura 5.14: Velocidad del viento del segundo test a 500 m de altura

Como test para un problema de emisión de contaminantes de una planta energética localizada en la isla de La Palma, se ha añadido una chimenea a la geometría original. Se considerará una chimenea de 200 m de altura, con un diámetro de 40 m en la base y 20 m en lo alto (figuras 5.15 y 5.16).

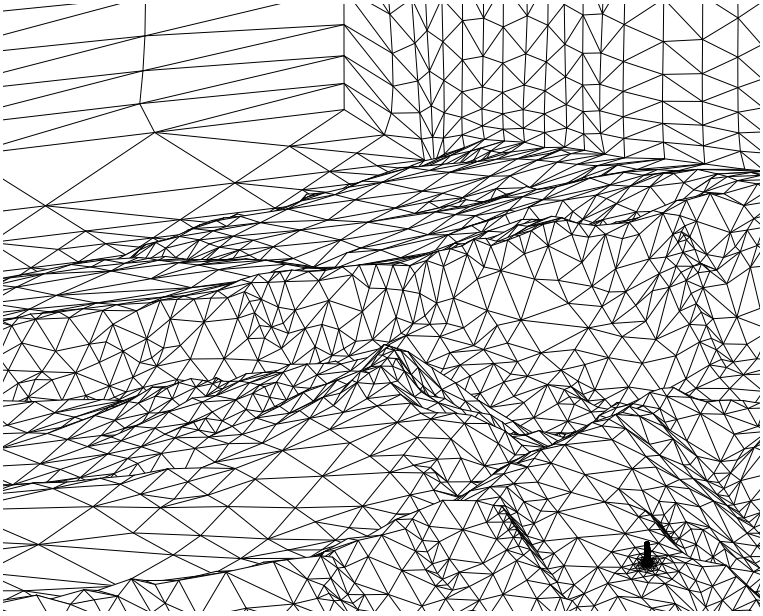


Figura 5.15: Zoom de la orografía con la chimenea cerca de la esquina inferior derecha

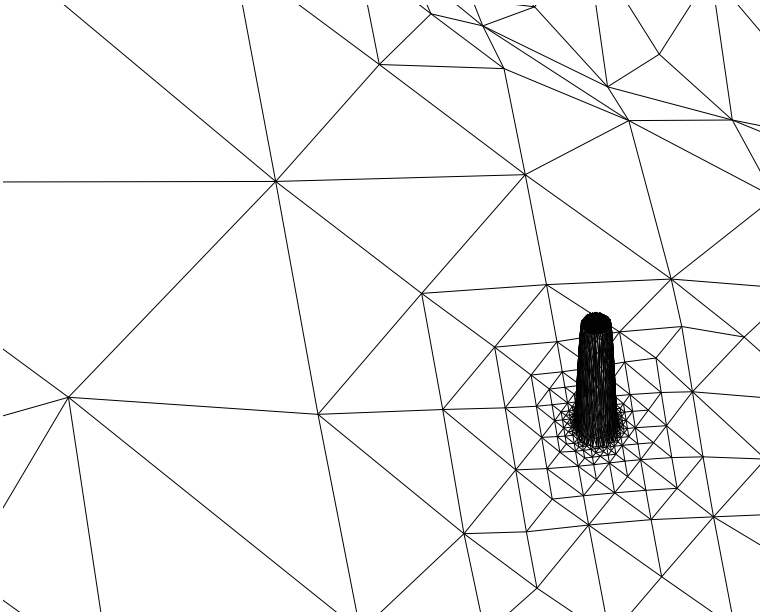


Figura 5.16: Detalle de la malla en torno a la chimenea

Se han aplicado seis etapas de refinamiento local en la trayectoria de la pluma para obtener una malla final de 31555 nodos y 170784 tetraedros. En la figura 5.17 se muestra un detalle del campo de viento ajustado con el efecto

causado por la introducción de la chimenea.

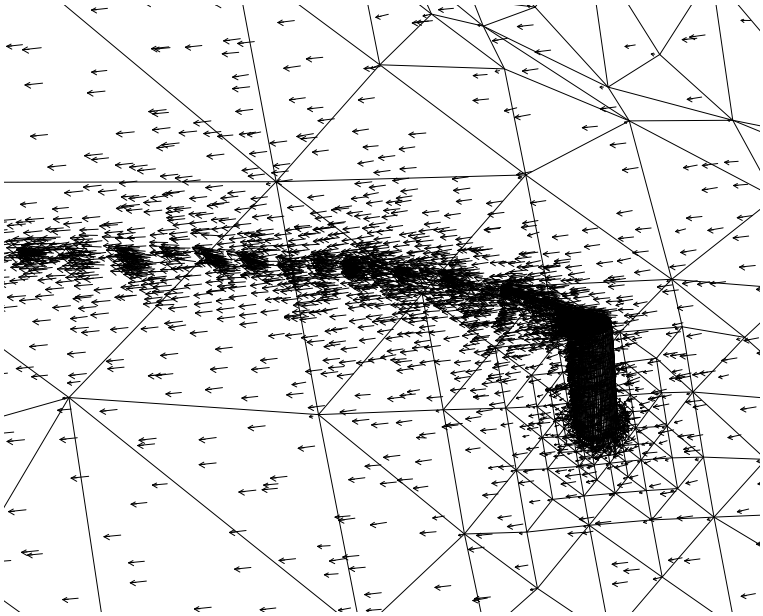


Figura 5.17: Velocidad del viento para la figura 5.16

5.4. Refinamiento Global vs. Local

5.4.1. Introducción

En la presentación del algoritmo de refinamiento se comentaba la necesidad de disponer de un estimador de error o de algún tipo de indicador que señalara qué elementos de la malla debían ser refinados para ajustar la solución numérica. Estos estimadores de error, dependiendo del tipo de problema, podrían ser difíciles de encontrar. En algún caso, incluso sería imposible obtener uno realmente fiable.

Sin embargo, en el algoritmo de desrefinamiento se utiliza la solución numérica del problema más un parámetro ϵ como herramientas para llevar a cabo el marcado de los elementos a eliminar. La solución numérica siempre estará disponible, y el parámetro está definido por el usuario, correspondiéndose con la precisión deseada para dicha solución.

En la aplicación presentada ([González-Yuste et al., 2006]) se propone emplear refinamiento global en lugar de refinamientos locales. Con el refinamiento global no se necesitan estimadores de error, puesto que toda la malla es refina-

da. Un proceso posterior de desrefinamiento eliminará aquellos elementos en los que la solución numérica no es significativa.

Cada iteración de este método conlleva un coste computacional mayor que con el refinamiento local, pero el número total de iteraciones podría ser mucho menor si no se escoge una buena estrategia de refinamiento. Además, no se necesita disponer de un estimador de error, y el método quedaría soportado únicamente por parámetros numéricos.

Tanto para la fase de implementación como para las aplicaciones de ha empleado el modelo de viento anteriormente expuesto.

5.4.2. Implementación

El desarrollo básico consiste en obtener una serie de mallas hasta llegar a una malla objetivo que cumpla con el criterio en la solución impuesto por el parámetro de desrefinamiento. El criterio de parada se establecerá mediante la comparación de algún parámetro de la malla obtenida con la anterior.

En este trabajo se ha empleado el número de nodos como parámetro a comparar, ya que expresa el número de puntos en los que se obtiene una solución numérica. Se ha definido w_n como el número de nodos de la malla T_n.

La primera implementación puede verse en el algoritmo 5.2. En cada iteración se realiza un refinamiento global, se calcula la solución numérica y se desrefina. El criterio de parada se ha establecido cuando dos mallas consecutivas tienen el mismo número de nodos, es decir, todo lo añadido en el paso anterior ha sido eliminado.

Algoritmo 5.2 Aproximación inicial del refinamiento global-desrefinamiento

T_0 malla inicial
n=0
repetir
    n=n+1
    $T'_n = T_{n-1}$ refinada globalmente
    Calcular solución numérica de T'_n
    $T_n = T'_n$ desrefinada según parámetro ϵ
hasta $w_n = w_{n-1}$

En la figura 5.18 se muestra la evolución de w_n esperada para el proceso. Debería haber un incremento considerable en el número de nodos tras el primer refinamiento, para ir estabilizando el crecimiento a medida que pasaran las

etapas. Finalmente, dejaría de crecer, momento en el que se considera finalizadas las iteraciones.

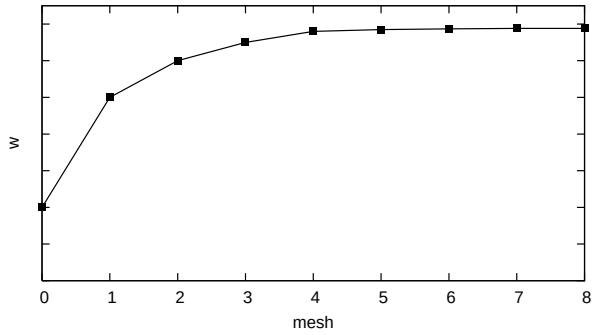


Figura 5.18: Evolución esperada de w

En los primeros tests que se realizaron se obtuvieron buenos resultados con valores de ϵ relativamente altos ($6m/s$). Pero con valores más bajos de ϵ ($2m/s$) el algoritmo no funcionó adecuadamente. En la figura 5.19 puede verse el comportamiento de w para dicho test.

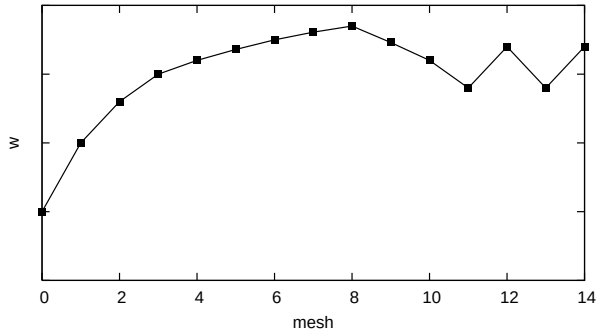


Figura 5.19: Evolución de w para $\epsilon = 2m/s$

Se pueden observar dos aspectos:

1. Hay mallas con $w_k < w_{k-1}$. El refinamiento mejora la solución numérica en algunas zonas y el desrefinamiento suprime elementos introducidos algunos pasos antes.

2. Hay dos grupos de mallas:

   - $w_{11} = w_{13} = w_{15} = \ldots$

- $w_{12} = w_{14} = w_{16} = \ldots$

El segundo punto implica que la condición del algoritmo 5.2, $w_n = w_{n-1}$, no se podrá alcanzar nunca. Para resolver esta situación se ha redefinido la comparación entre mallas. Se considerarán dos mallas T_k y T_n $(k < n)$ son similares cuando la diferencia entre w_k y w_n es menor que un porcentaje definido por el usuario. Lo que se va a realizar, en definitiva, es comparar las w de las mallas con todas las anteriores.

En el test final, con $\epsilon = 1.5m/s$ se obtuvo un comportamiento como el de la figura 5.20. La solución numérica nunca se podría ajustar en estos casos, pero es un problema particular del modelo de viento empleado, en aquellos elementos muy cercanos al terrero. La diferencia entre la solución numérica en elementos de tierra y sus adyacentes presenta muy bajas variaciones con el refinamiento. Para los nuevos elementos introducidos en esa zona, la solución numérica es similar a la de los anteriores, por lo que no es mejorada. El desrefinamiento no elimina esos elementos, por lo que volverán a ser refinados una y otra vez.

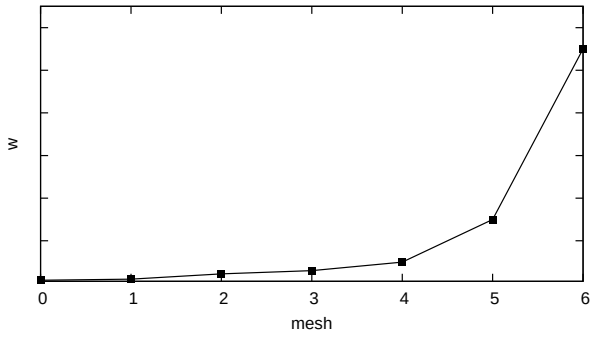


Figura 5.20: Evolución de w para $\epsilon = 1.5m/s$

Para evitar este comportamiento se ha introducido un nuevo parámetro δ. Indica el tamaño mínimo de una arista en cualquier malla. El proceso de desrefinamiento se hará atendiendo a dos parámetros: ϵ y *delta*, haciendo que cualquier elemento que tenga una arista menor que δ sea eliminado. La implementación definitiva puede verse en el algoritmo 5.3.

5.4.3. Aplicaciones

Todas las implementaciones de ejecutaron en un XEON con procesador dual, 2 Gb de RAM. con linux como S.O. y programas compilados con GNU C++.

Algoritmo 5.3 Implementación definitiva del refinamiento global - desrefinamiento

T_0 malla inicial
n=0
loop
  n=n+1
  $T'_n = T_{n-1}$ refinada globalmente
  Calcular solución numérica de T'_n
  $T_n = T'_n$ desrefinada según parámetros ϵ y δ
  para i=0 to n-1
    fin si $w_n \approx w_i$
  fin para
fin loop

Para detener el proceso de ha establecido que dos mallas serán similares si sus w_i son menores que el 1 %:

$$\frac{w_k}{w_n} \in [0.99, 1.01], k < n \tag{5.27}$$

La primera malla empleada en un curva de gauss en 3D mostrada en la figura 5.21. Consta de 1680 nodos and 7645 tetraedros para un dominio simulado de $10000 \times 10000 \times 10000\ m^3$.

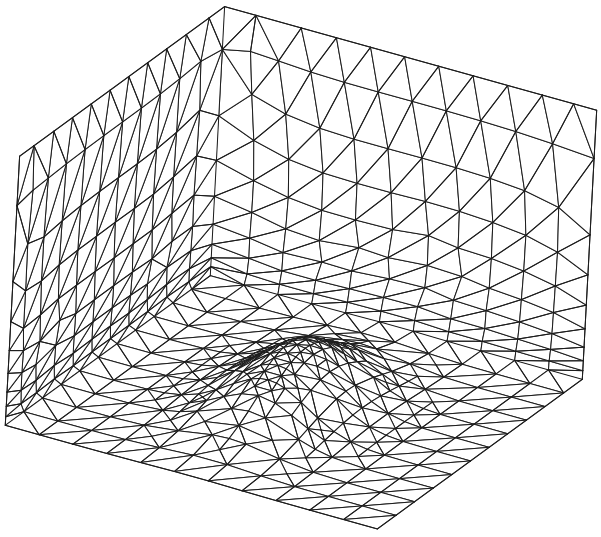


Figura 5.21: T_0: curva de gauss en 3D de $10000 \times 10000 \times 10000\ m^3$

El primer test fue realizado con los parámetros $\epsilon = 2m/s$ y $\delta = 40m$. En cinco pasos se consiguió ajustar la malla (se muestran la resultante en fig:glo-

loc-t0-2-40).

En la figura 5.2(a) se puede ver un gráfico de la evolución de w y en la tabla 5.2(b) se observan los tiempos de CPU de cada proceso.

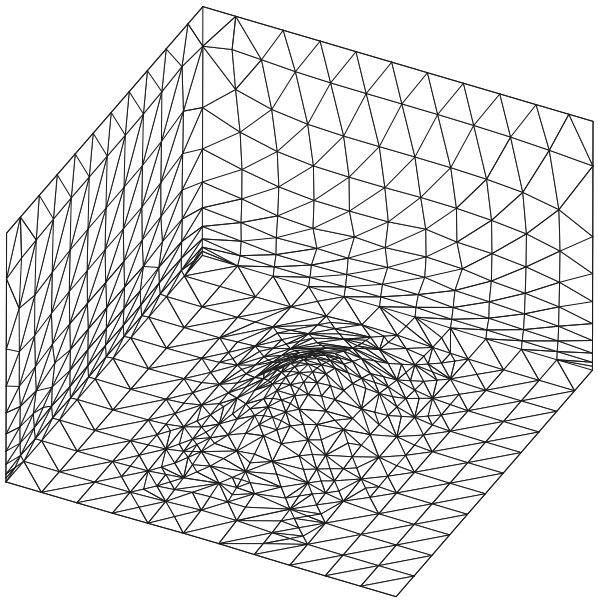
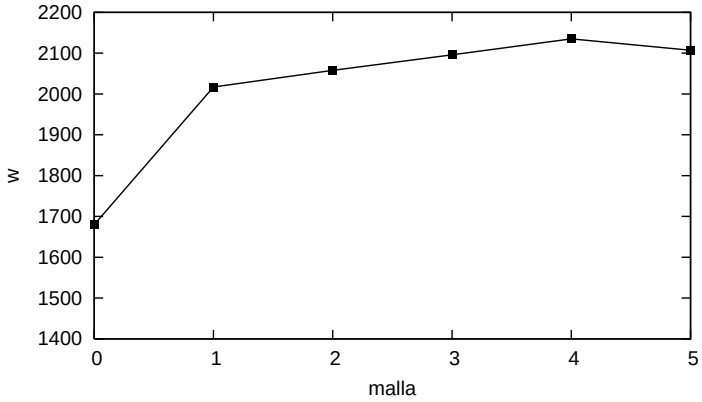


Figura 5.22: Malla T_5 obtenidas de T_0 con $\epsilon = 2m/s$ y $\delta = 40m$

(a) Evolución de w

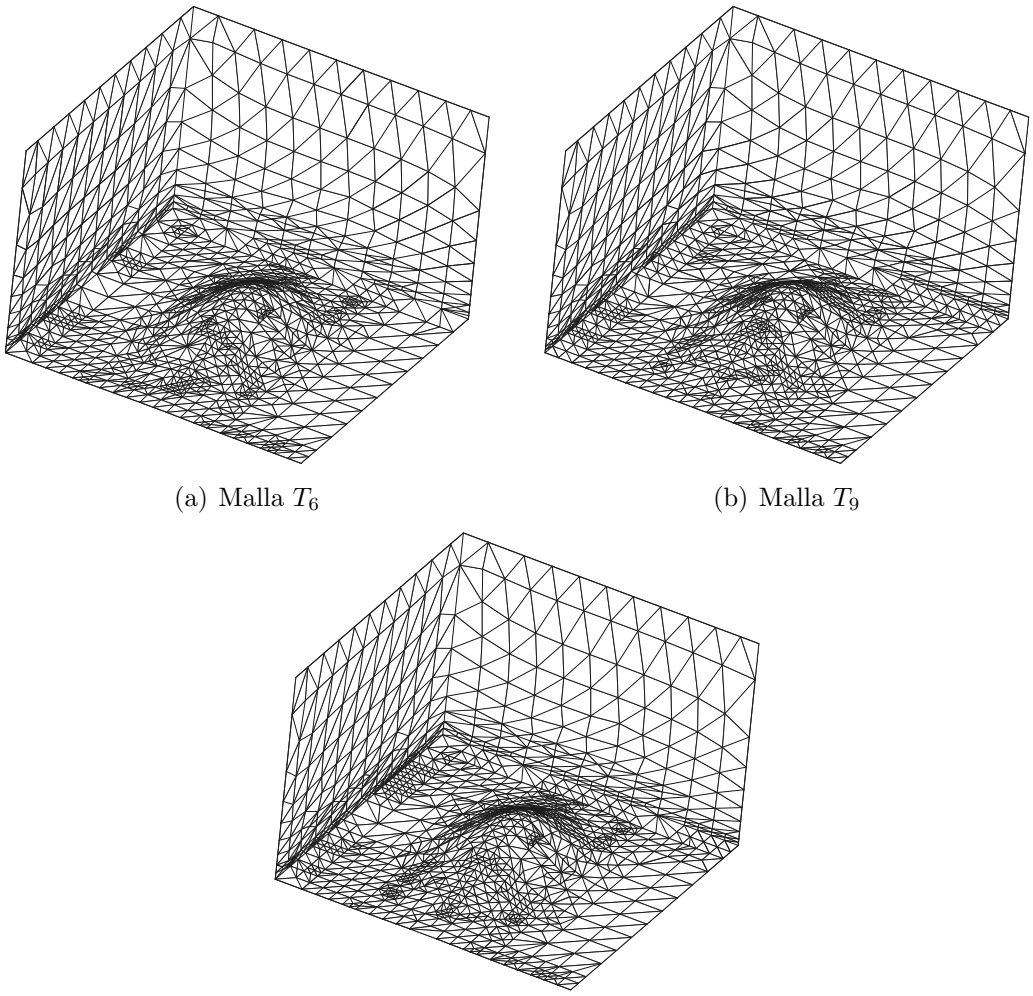


(b) Tiempo de CPU en segundos

Inicial T_{n-1}	w_{n-1}	Refin.	$w_n^{'}$	Sol.Numer.	Desref.	Final T_n	w_n
T_0	1680	3.72	11787	5.93	2.68	T_1	2017
T_1	2017	5.18	12591	5.31	3.16	T_2	2058
T_2	2058	5.39	12865	5.52	3.45	T_3	2096
T_3	2096	5.76	13084	5.51	3.83	T_4	2135
T_4	2135	6.09	13326	5.98	4.00	T_5	2107

Tabla 5.3: Datos para T_0 con $\epsilon = 2m/s$ y $\delta = 40m$ (Figura 5.22)

Se realizó otro test con parámetros $\epsilon = 1.5m/s$ y $\delta = 80m$. Se consiguió una malla ajustada en 12 pasos (figura 5.23 y tabla 5.4).



(a) Malla T_6

(b) Malla T_9

(c) Malla T_{12}

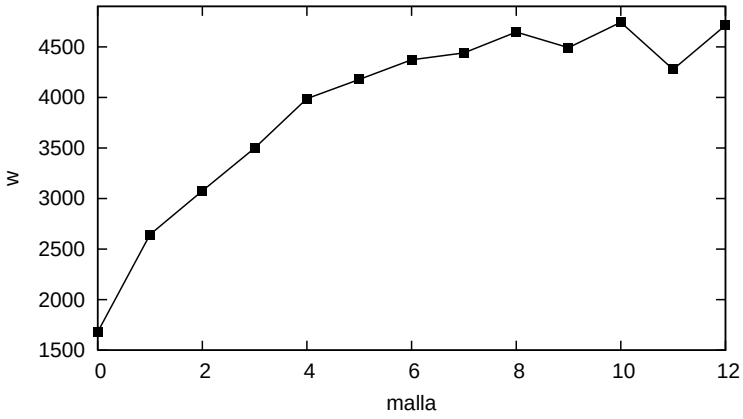
Figura 5.23: Mallas obtenidas de T_0 con $\epsilon = 1.5m/s$ y $\delta = 50m$

En ambos test el algoritmo funcionó correctamente y fueron útiles para ajustar y validar la implementación.

Se ha usado une geometría real generada con [Escobar et al., 2003; Montenegro et al., 2002b] (figura 5.24). Representa una parte del sur de la isla de La Palma, y consta de 4535 nodos y 21137 tetraedros de un dominio real de $45600 \times 31200 \times 6000\ m^3$.

El objetivo en este caso es obtener una malla ajustada a $\epsilon = 4m/s$ y $\delta = 40m$. Con solo cinco pasos se logró. En la figura 5.25 se pueden ver algunas mallas generadas y en la tabla 5.5 información del proceso.

(a) Evolución de w



(b) Tiempo de CPU en segundos

Inicial T_{n-1}	w_{n-1}	Refin.	$w_n^{'}$	Sol.Numer.	Desref.	Final T_n	w_n
T_0	1680	3.78	11787	6.01	2.94	T_1	2644
T_1	2644	6.84	15857	7.43	4.57	T_2	3075
T_2	3075	8.32	18775	9.69	7.05	T_3	3498
T_3	3498	9.82	21564	11.67	8.02	T_4	3987
T_4	3987	11.73	24532	14.40	8.15	T_5	4178
T_5	4178	12.26	25982	16.70	10.46	T_6	4372
T_6	4372	13.36	27250	16.85	10.77	T_7	4440
T_7	4440	13.96	27597	18.23	11.43	T_8	4647
T_8	4647	14.64	29158	20.26	12.35	T_9	4492
T_9	4492	14.76	28123	18.52	11.49	T_{10}	4743
T_{10}	4743	16.12	29617	18.55	12.43	T_{11}	4277
T_{11}	4277	14.60	26783	16.73	11.61	T_{12}	4713

Tabla 5.4: Datos para T_0 con $\epsilon = 1.5m/s$ y $\delta = 80m$ (figura 5.23)

Por otro lado se ha usado la misma malla lp_0 con el método tradicional, es decir, refinamiento local - estimación de error - desrefinamiento. El parámetro γ para el refinamiento se ajustó a 0,6 y el parámetro de desrefinamiento ϵ tomó el valor de la anterior ejecución ($4m/s$). El parámetro δ no se empleó. Las mallas que se han obtenido son similares a las que se pueden ver en la figura 5.25, pero en la tabla 5.6 y en la figura 5.26 se puede observar que el número de pasos es mayor.

5.4.4. Conclusión

Se ha visto, anteriormente, la necesidad de disponer de indicadores de error para ajustar mallas mediante refinamientos locales, y no siempre están disponi-

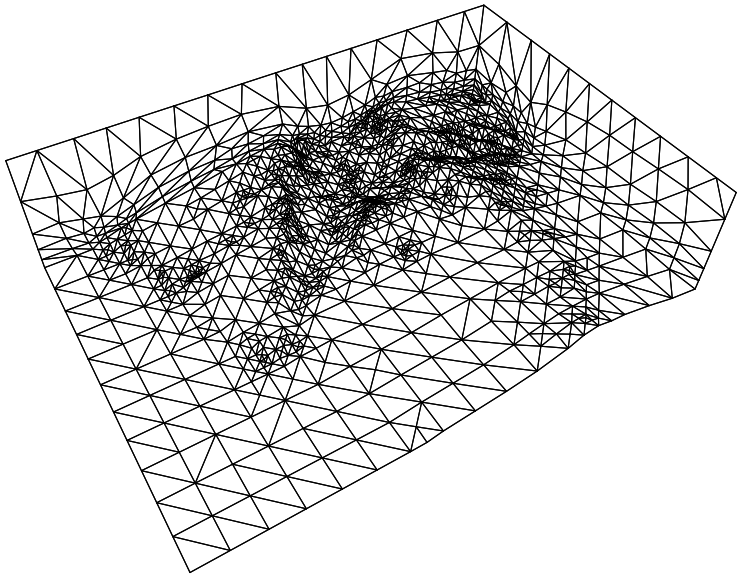
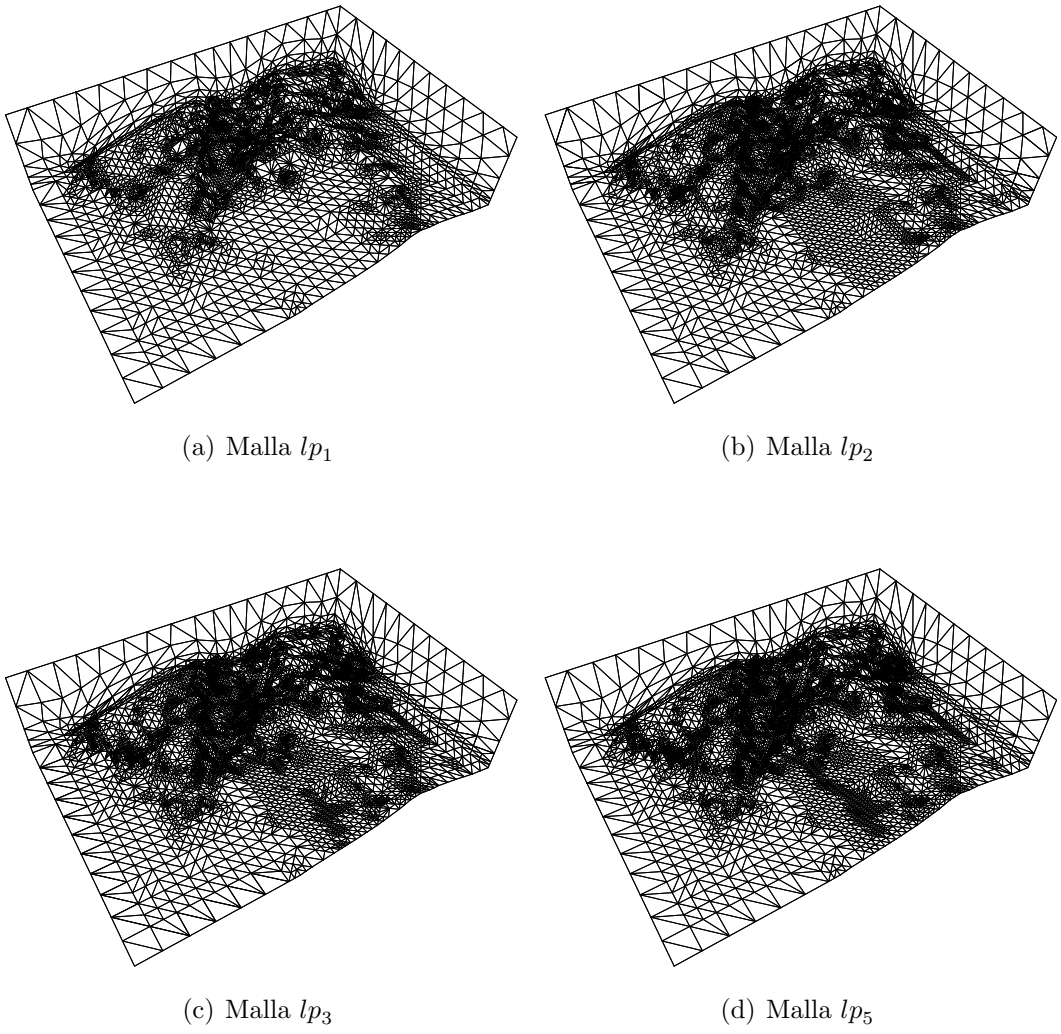


Figura 5.24: lp_0: Sur de la isla de La Palma de $45600 \times 31200 \times 6000\ m^3$

bles. El número de pasos de refinamiento depende tanto de la posición como de la cantidad de singularidades de la solución numérica. Es coste computacional, en cada paso, es menor que el método que se ha propuesto, básicamente porque el número de elementos procesados es menor.

Con el refinamiento global todas las singularidades se tratan simultáneamente. Se evita el uso de estimadores de error y el método solo dependería de parámetros numéricos: ϵ y, en el caso de esta aplicación, δ. Por supuesto, ya que toda la malla es procesada en cada paso, el coste el mayo, pero el número paso es menor.

Si el número de iteraciones es muy alto usando refinamiento local, este método podría se ventajoso, ya no solo porque sólo emplea únicamente parámetros numéricos, sino en tiempos de CPU.



(a) Malla lp_1

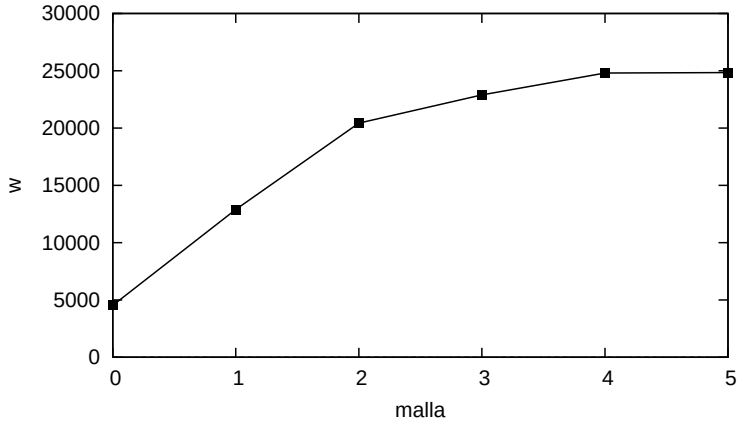
(b) Malla lp_2

(c) Malla lp_3

(d) Malla lp_5

Figura 5.25: Mallas obtenidas de lp_0 con $\epsilon = 4m/s$ y $\delta = 40m$

(a) Evolución de w



(b) Tiempo de CPU en segundos

Inicial T_{n-1}	w_{n-1}	Refin.	$w_n^{'}$	Sol.Numer.	Desref.	Final T_n	w_n
lp_0	4535	9.04	32139	23.16	8.37	lp_1	12898
lp_1	12898	29.83	83648	99.84	27.71	lp_2	20426
lp_2	20426	46.58	125989	177.27	42.19	lp_3	22887
lp_3	22887	52.47	139159	211.89	53.05	lp_4	24806
lp_4	24806	57.92	150243	203.95	59.67	lp_5	24845

Tabla 5.5: Datos para lp_0 con $\epsilon = 4m/s$ y $\delta = 40m$ (figura 5.25)

Inicial T_{n-1}	w_{n-1}	Refin.	$w_n^{'}$	Sol.Numer.	Desref.	Final T_n	w_n
lp_0	4535	5.90	18200	10.59	5.03	lp_1	12768
lp_1	12768	19.75	44340	35.01	19.00	lp_2	17593
lp_2	17593	28.38	45780	39.78	19.62	lp_3	18773
lp_3	18773	37.23	42330	33.26	22.35	lp_4	19108
lp_4	19108	38.88	41944	31.12	23.72	lp_5	19304
lp_5	19304	45.69	49265	44.49	27.29	lp_6	19519
lp_6	19519	48.27	52984	50.30	29.66	lp_7	20058
lp_7	20058	56.93	55388	57.96	31.93	lp_8	20294
lp_8	20294	59.61	54401	45.93	32.72	lp_9	20536
lp_9	20536	52.58	51416	49.49	32.04	lp_{10}	20505

Tabla 5.6: Datos para lp_0 con $\gamma = 0.6$ y $\epsilon = 4m/s$

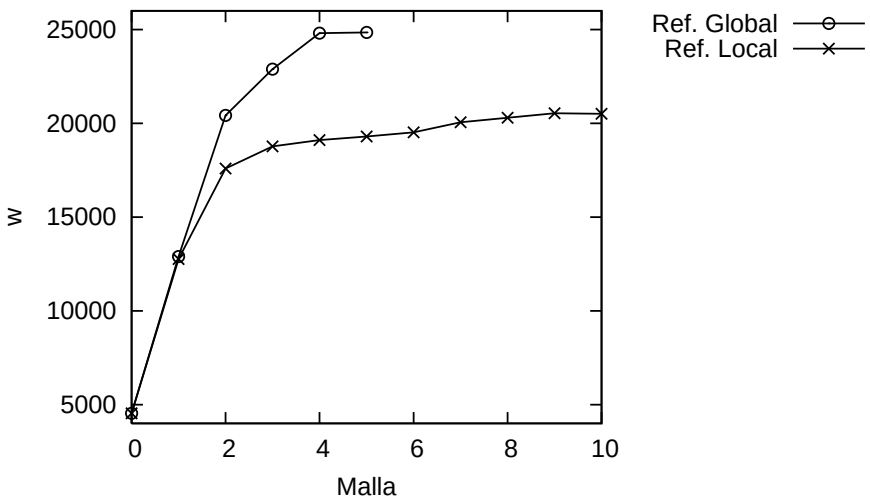


Figura 5.26: Evolución de w ajustando lp_0 al usar refinamiento global (tabla 5.5) y local (tabla 5.6)

Capítulo 6

Conclusiones y líneas futuras

6.1. Líneas futuras

Hay una serie de cuestiones que a la conclusión de este trabajo aún no han sido completadas. Si bien el objetivo primordial era una implementación efectiva de los algoritmos, no deja de ser una cuestión abierta su uso en otras funciones o la mejora en la implementación de los mismos.

En los siguientes puntos se comentan algunas aplicaciones que, aunque no se puede decir que están operativas, sí se ha obtenido algún resultado que invita a continuar trabajando en esa línea.

Por otra parte, la paralelización de los algoritmos es una tarea muy importante. Aunque el algoritmo de refinamiento puede operar en múltiples procesadores, en las siguientes secciones se va más allá, proponiendo un modelo de paralelización en máquinas totalmente independientes pero trabajando de forma cooperativa. En este caso se cuenta con un diseño completo sobre cómo debe ser el esquema, pero la implementación aún está en una fase bastante temprana.

6.1.1. Mallador en 3 dimensiones

En [Montenegro et al., 2002b] se propone un generador de mallas en tres dimensiones para orografías. Partiendo de una digitalización del terreno se construye una malla de triángulos en dos dimensiones que, tras sucesivas etapas de refinamiento global con el algoritmo 4T de Rivara, sea capaz de capturar toda la información de la topografía.

Una vez obtenida dicha malla, se aplica el desrefinamiento para eliminar aquellos nodos que no aporten información relevante del dominio. Para generar

los puntos sobre la superficie del terreno se empleará una función de espaciado vertical que indicará la distribución de los nodos, apoyándose en un paralelepípedo que marcará el contorno del volumen tridimensional. Para unir este conjunto de puntos se emplea una variante de la triangulación de Delaunay [Escobar y Montenegro, 1996]. Finalmente, la base del paralelepípedo, que contendrá la digitalización de la orografía deberá ser comprimida en función de la altura de cada punto.

Debido a este último proceso de compresión es posible que se produzcan enredos en la malla, o cuanto menos, una degeneración de los tetraedros que la componen. Será necesario, por tanto, aplicar técnicas de desenredo y suavizado para obtener una malla válida a efectos de procesamiento.

6.1.1.1. Uso de algoritmos en 3D

Se han realizado algunas pruebas para obtener una malla adaptada a una orografía a usando un esquema más simple con los algoritmos de refinamiento/desrefinamiento propuestos en este trabajo. El esquema es similar, pero al partir de un elemento tridimensional no será necesario realizar la distribución de puntos sobre la superficie del terreno.

Partiendo de un paralelepípedo dividido en un cierto número de tetraedros se realizan refinamientos globales de los elementos que se encuentran en la base del mismo. El número de etapas deberá ser tal que se capture toda la información de la topografía digitalizada. Así se dispondría de un geometría tridimensional con una nube de puntos conectada en su interior.

A continuación se aplica el algoritmo de desrefinamiento, eliminando aquellos elementos que no aporte información relevante sobre la orografía. Se realizaría la compresión de la base del paralelepípedo y, simultáneamente, de todos los puntos sobre la misma de forma proporcional a la altura a la que están sobre el terreno.

Esta forma de proceder disminuye considerablemente el número de enredos que se producen en la malla obtenida, a la vez que mejora la calida de los tetraedros resultantes. Aplicando un algoritmo de suavizado permitirá obtener una malla final adecuada para su uso posterior.

6.1.1.2. Tratamiento inicial

Una cuestión que se ha observado fundamental es la forma y la división del paralelepípedo inicial. Cuando se trata de mallas un dominio tridimensional de forma aproximada a un cubo, es suficiente partir de un solo elemento dividido en tetraedros, puesto que estos últimos tendrán una calidad inicial aceptable, lo que a su vez generarán nuevos elementos de buena calidad. Pero cuando se trata de dominios con medidas muy diferentes, lo ideal es tener otro tipo de disposición inicial que a su vez genere tetraedros de buena calidad.

Lo que se plantea es dividir un paralelepípedo inicial en elementos con formas más regulares, formas cercanas a cubos. A su vez, se debe realizar una partición de dichos cubos optimizada en cuanto a número de elementos generados. Se han realizado algunas pruebas para automatizar dicho proceso a partir del máximo común divisor de las medidas de la topografía, empleando redondeos para evitar elementos degenerados. Si bien no se disponen de resultados concluyentes, los obtenidos hasta ahora parecen indicar una buena opción.

Por otra parte se plantea la división inicial de los paralelepípedos en tetraedros. Los primeros test se han realizado dividiéndolos en cinco tetraedros pero se plantea la opción de dividirlo en seis tetraedro. Aunque la primera posibilidad generaría un número menor de tetraedros en la malla final, sería conveniente realizar un estudio de calidad para determinar qué opción sería más viable. Este planteamiento podría ser incluso dependiente del problema en cuestión.

6.1.1.3. Estructuras de datos

Todo el proceso parte de una malla inicial de cubos, originados de la partición de un paralelepípedo inicial. Éste podría verse como una malla de un sólo elemento. Se tiene, en definitiva, una malla que generará, mediante división de elementos, una nueva malla base para obtener los tetraedros iniciales.

Para contener la estructura de la malla de cubos se plantea la posibilidad de emplear una jerarquía similar a la empleada en la malla de tetraedros (figura 2.5). La diferencia estribaría en que se usarían caras de 4 aristas y elementos formados por 6 caras.

La solución adoptada, en fase de implementación, es la generación de una súper-estructura de datos basada en plantillas de *C++* capaz de tratar ambos tipos de malla como casos particulares. Esta nueva estructura contendría los métodos comunes a cualquier definición de malla, mientras que las instancias

que detallan cada tipo de malla contendrían los métodos específicos.

6.1.2. Desrefinamiento de mallas de triángulos

En la sección 5.1 se presentó una adaptación del algoritmo de refinamiento para trabajar con mallas de triángulos. Esta implementación resultó sencilla puesto que lo único que hubo de hacerse fue programar las singularidades del algoritmo a nivel de cara, dejando de lado el procesamiento de tetraedros.

Una adaptación que ha quedado pendiente es la del algoritmo de desrefinamiento. El planteamiento de base es similar: marcar todos los nodos como desrefinables y desmarcar aquellos que no cumplan la condición de desrefinamiento. A partir de aquí podría plantearse la eliminación de los elementos marcados para desrefinar. No habría que realizar estudios por nivel para determinar si un elemento debe quedarse por conformidad puesto que están definidos los posibles tipos de división en función del número de marcas de cada cara.

Con esta implementación quedaría completo el par de algoritmos para el tratamiento de mallas de triángulos, con lo que se tendría una herramienta útil para el proceso de las mismas.

6.1.3. Paralelización del algoritmo de refinamiento

Tal y como se ha descrito en la sección 3.4.1, el algoritmo de refinamiento está programado con un cierto grado de paralelización. Este procesamiento en paralelo está enmarcado en *threads*, o hilos de ejecución. La característica principal es que comparten espacio de memoria, si bien pueden estarse ejecutando en diferentes CPU. Es un sistema muy simple de paralelización puesto que no lleva asociado más que sincronismo de secciones críticas.

La propuesta de futuro va más allá. Se pretende realizar un diseño e implementación de un procesamiento de mallas totalmente distribuido, de forma que en varias máquinas independientes se puedan realizar tareas de refinamiento.

La idea, en líneas generales, consiste en disponer de una máquina principal (servidor) encargada de leer la malla inicial. Esta máquina partiría en n trozos dicha malla, que serían enviados a n máquinas independientes (clientes). El servidor indicaría a cada cliente qué elementos debe refinar y cada uno de estos realizaría las particiones necesarias e informaría al servidor de los elementos resultantes (figura 6.1).

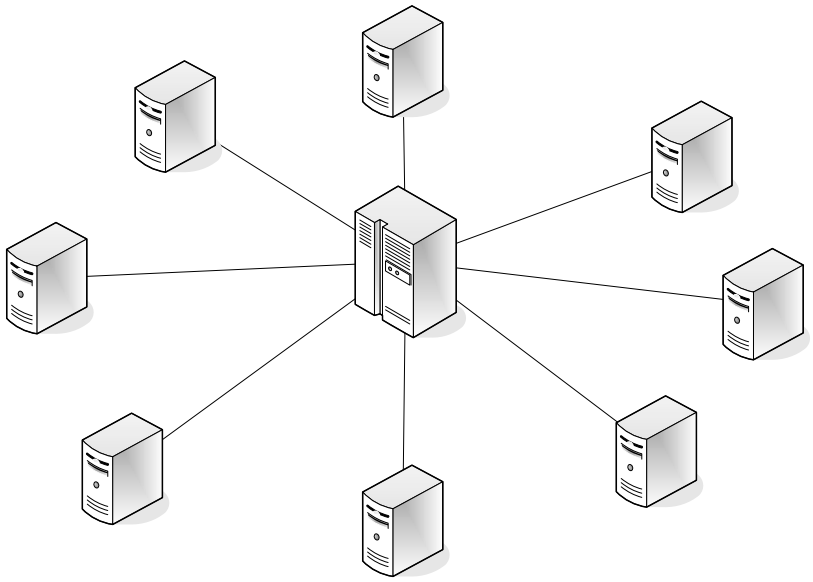


Figura 6.1: Equipos en procesamiento distribuido

Como se ha mencionado, el primer paso es realizar una partición de la malla inicial de varios trozos. Por lo tanto, existirán una serie de elementos *frontera* entre las diferentes partes de la malla. Estos elementos frontera serán caras, arista y nodos, y estarán contenidos en dos clientes diferentes, por lo que cualquier cambio en ellos (marcado o división) tendrá que verse reflejados en todos ellos. Deberá existir un proceso de comunicación entre clientes para realizar este sincronismo.

Se plantea en primer lugar disponer de un algoritmo que optimice el número de elementos frontera a la hora de dividir una malla y así minimizar el trasiego de información entre clientes. Para realizar la sincronización se necesita disponer de una identificación única de los elementos, de manera que haya una referencia unívoca sobre cual se realiza una operación. Se disponen de dos opciones para realizar la sincronización:

- Cuando una máquina cliente realiza alguna operación sobre un elemento frontera se informa a la otra máquina. Esto implica que cada cliente debe conocer en cual está el elemento, por lo que si se disponen de n máquinas se podrían establecer $n \times n$ canales de comunicación.

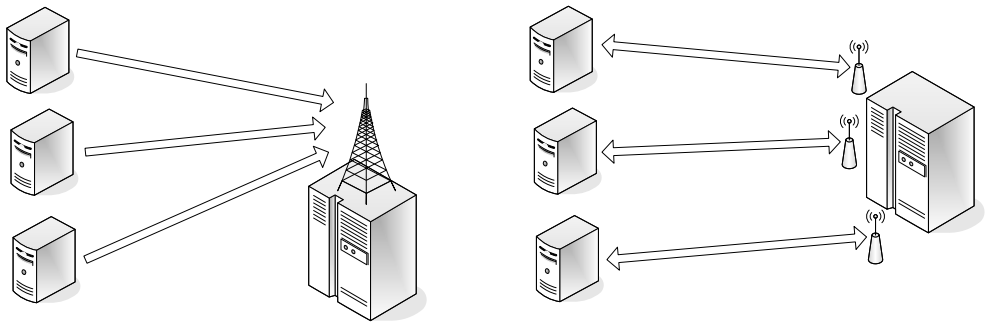
- Todas las comunicaciones pasan a través del servidor. Puesto que ésta sabe a dónde han ido a parar cada elemento, cuando se realice alguna operación, el cliente implicado informará a la principal, que hará lo propio

con la otra máquina. Esta opción disminuye los canales de comunicación, pero aumenta el trabajo del servidor.

En ambos casos se puede ver la necesidad de tener el mínimo número de elementos frontera, aunque la segunda opción parece la más viable pues simplifica el trabajo de los clientes que, en definitiva, es el objetivo.

Algunas pruebas se han realizado con una implementación básica del esquema. En una intranet se han definido varias máquinas receptoras y una principal comunicadas a través de *sockets*. Los *sockets* permiten el intercambio de un flujo de datos entre aplicaciones sobre una red mediante un protocolo definido, una dirección IP y un número de puerto. Se ha elegido el protocolo TCP por estar ampliamente extendido y cumplir una serie de características, como que es orientado a conexión y garantiza que toda la información llega al destino sin errores y en el mismo orden en que fue enviada.

El servidor lanza un proceso de escucha (*listener*) sobre un número de puerto determinado (figura 6.2(a)). Cuando un cliente se inicia realiza un proceso de conexión con el servidor. En este momento, el servidor pasa la comunicación a otro *listener* dedicado exclusivamente a comunicarse con el cliente (figura 6.2(b)). Una vez se hayan recibido todas las conexiones esperadas podrá comenzar el tratamiento de la malla propiamente dicho.



(a) Inicial (b) Dedicada

Figura 6.2: Comunicación en cliente/servidor

Toda la implementación de los *listener*, tanto en el servidor como en los clientes se ha realizado empleando *sockets* sobre *threads* para permitir el tratamiento de varias comunicaciones simultáneamente. Además, se han implementado una serie de rutinas para determinar que no se ha perdido la comunicación con el cliente (y viceversa). Y este es uno de los puntos débiles del esquema: su nula

capacidad de recuperación ante una caída. En caso de producirse la caída de un cliente o la pérdida de comunicación con el mismo se deberá abortar todo el proceso.

Sin embargo, una de las ventajas de este esquema es su escalabilidad. Realizando módulos que hagan el papel de cliente/servidor de forma simultánea se podría lograr un esquema como el mostrado en la figura 6.3. Es este caso de dispone de un gran servidor que estaría comunicado con clientes individuales y con otros servidores que dispondrían de sus propios clientes. Estos servidores actuarían como clientes respecto del servidor principal. Es estos caso se debería replantear la división de la malla en función de la carga de trabajo admisible por cada máquina o grupo máquinas.

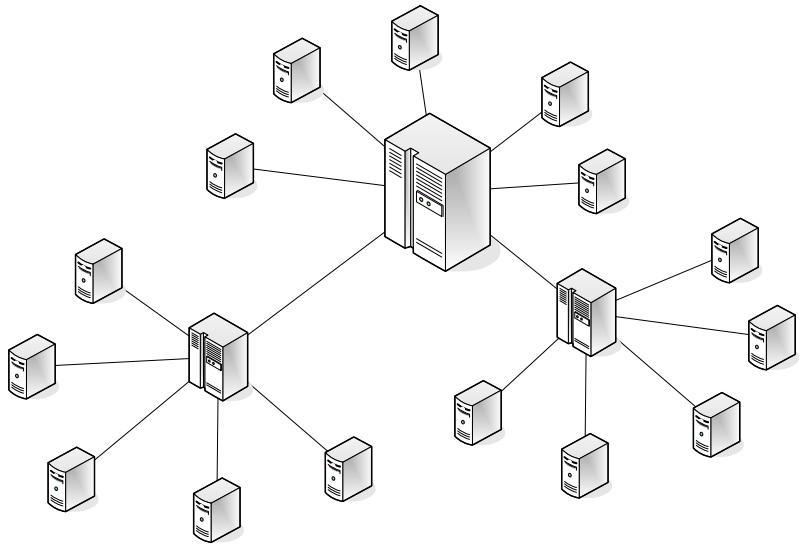


Figura 6.3: Esquema en cliente/servidor escalable

6.1.4. Paralelización del algoritmo de desrefinamiento

En el caso del algoritmo de desrefinamiento no se ha realizado ningún tipo de esquema para la paralelización del mismo.

A un primer nivel, paralelización por *thread*, habría que analizar las posibles secciones críticas que hubieran en el proceso. En principio podrían darse en los lugares en que se hagan modificaciones sobre los nodos de los tetraedros, desmarcándolos como desrefinables (algoritmo 4.3).

El recorrido de listas por nivel podría lanzarse en paralelo. Para ello habría

que emplear una estructura de datos que permitiera acceso concurrentes y mutuamente excluyentes, como la clase *IBuffer*. Del mismo modo, se podrían realizar, simultáneamente recorridos por todos los niveles, puesto que una operación sobre un tetraedro de un cierto nivel de profundidad afectará a los resultados de los niveles anteriores.

Si el proceso de división del algoritmo de refinamiento fue posible paralelizarlo, a priori, el proceso de eliminación de elementos también. Es una acción que se realiza en cada tetraedro de forma independiente, por lo que no habría conflictos.

Finalmente, habría que estudiar detalladamente la forma de incluir este algoritmo en el esquema propuesto en el punto anterior, en el tratamiento distribuido de la malla entre diferentes máquinas.

6.2. Conclusiones

En el presente trabajo se ha intentado plasmar una visión estructurada de la implementación de los algoritmos de refinamiento/desrefinamiento desde su análisis y su desarrollo.

Queda una interesante línea de investigación abierta en cuanto a la generación de un modelo de datos capaz de asimilar mallas no sólo de elementos uniformes, sino de diversos tipos de elementos. Esto permitiría dar aproximaciones en bruto de formas que, mediante refinamientos adaptativos, ajustaran una geometría compleja.

La modelización de los elementos de la malla, haciendo que cada uno sea altamente independiente, ha permitido la paralelización de sus procesos críticos. Además, las vinculaciones entre elementos mediante referencias simples ha facilitado la implementación de los módulos asociados al tratamiento de los mismos.

Alguna estructura presentada, como la cola con acceso concurrente llamada *IBuffer*, está siendo una herramienta fundamental en aplicaciones en desarrollo, ya que permite que múltiples procesos accedan a datos de forma totalmente independiente y sin ningún tipo de conflictos.

Si bien los algoritmos han cumplido con las expectativas propuestas, los diseños en cuanto a la estructura de datos empleada han permitido su uso en aplicaciones para las que no estaban pensados, lo cual da idea de su versatilidad.

Referencias

Arnold, D., Mukherjee, A. y Pouly, L. Locally adapted tetrahedral meshes using bisection. *SIAM Journal on Scientific Computing*, 22(2): 431–448, 2001.

Bank, R., Sherman, A. y Weiser, A. Refinement algorithms and data structures for regular local mesh refinement. *Scientific Computing*, páginas 3–17, 1983.

Bank, R. y Smith, R. Mesh smoothing using A posteriori error estimates. *SIAM Journal on Numerical Analysis*, 34(3): 979–997, 1997.

Barnard, J., Wegley, H. y Hiester, T. Improving the performance of mass consistent numerical model using optimization techniques. *Journal of Applied Meteorology*, 26(26): 675–686, 1987.

Bazaraa, M., Sherali, H. y Shetty, C. *Nonlinear Programming: Theory and Algorithms*. John Wiley and Sons Inc., 1993.

Bornemann, F., Erdmann, B. y Kornhuber, R. Adaptive multilevel methods in three space dimensions. *International Journal for Numerical Methods in Engineering*, 36: 3187–3203, 1993.

Boubel, R., Fox, D., Turner, D. y Stern, A. *Fundamentals of Air Pollution*. Academic Press, San Diego, 3ª edición, 1994.

Briggs, G. Plume rise. Informe técnico, United States Atomic Energy Commission Critical Review Series, Springfield, VA, 1969.

Briggs, G. Some recent analysis of plume rise observation. En H. Englund y W. Beery, editores, *Proceedinsg of the Second International Clean Air Congress*, páginas 1029–1032. Academic Press, Nueva York, 1971.

Briggs, G. Discussion: Chimney plumes in neutral and stable surrounding. *Atmospheric Environment*, 6: 507–510, 1972.

Briggs, G. Diffusion estimation for small emissions. Informe técnico, Atmosferic Turbulence and Diffusion Laboratory, Oak Ridge, TN, 1973.

Briggs, G. *Lectures on Air Pollution and Environmental Pollutants*, capítulo 3, páginas 59–111. American Meteorological Society, Boston, 1975.

Businger, J. y Arya, S. Heights of the mixed layer in the stable, stratified planetary boundary layer. *Advances in Geophysics*, 18A: 73–92, 1974.

Carey, G.F. *Computational Grids: Generations, Adaptation and Solution Strategies*. Series in Computational and Physical Processes in Mechanics and Thermal Sciences. Taylor & Francis, Washington, 1997.

Ceballos, F.J. *Enciclopedia del lenguaje C++*, capítulo 19: Hilos. Ra-Ma, Septiembre 2003.

Courant, R. Variational methods for solution of equilibrium and vibration. *Bulletin of the American Mathematical Society*, 49: 1–43, 1943.

Davis, C., Bunker, S. y Mutschlecner, J. Atmospheric transport models for complex terrain. *Journal of Climate and Applied Meteorology*, 23: 235–238, 1984.

de Baas, A. *Modelling of Atmospheric Flow Fields*, capítulo Scaling Parameters and their Estimation, páginas 87–102. World Scientific Publishing, Singapur, 1996.

De Jong, K. y Spears, W. A formal analysis of the role of multi-point crossover in genetic algorithms. *Annals of Mathematic and Artificial Intelligence*, 5(1): 1–26, 1992.

Djidjev, H. Force-directed methods for smoothing unstructured triangular and tetrahedral meshes. En *Proceedings of the 9th International Meshing Roundtable*, páginas 395–406, octubre 2000.

Dompierre, J., Labbé, J., Guibault, F. y Camarero, R. Proposal of benchmarks for 3d unstructured tetrahedral mesh optimization. En *Proceedings of the 7th International Meshing Roundtable*, páginas 459–478. Sandia National Laboratories, 1998.

Escobar, J. y Montenegro, R. Several aspects of three-dimensional delaunay triangulation. *Advances in Engineering Software*, 27(1-2): 27–39, 1996. ISSN 0965-9978.

Escobar, J., Montenegro, R., Montero, G. y Rodríguez, E. An algebraic method for smoothing surface triangulations on a local parametric space. *International Journal for Numerical Methods in Engineering*, 66: 740–760, 2006.

Escobar, J., Montenegro, R., Montero, G., Rodríguez, E. y González-Yuste, J. Smoothing and local refinement techniques for improving tetrahedral mesh quality. *Computers & Structures*, 83: 2423–2430, 2005.

Escobar, J., Rodríguez, E., Montenegro, R., Montero, G. y González-Yuste, J. Simultaneous untangling and smoothing of tetrahedral meshes. *Computer Methods in Applied Mechanics and Engineering*, 192(25): 2775–2787, 2003.

Ferragut, L., Montenegro, R. y Plaza, A. Efficient refinement/derefinement algorithm of nested meshes to solve evolution problems. *Comm. Numer. Methods Engrg*, 10: 403–412, 1994.

Freitag, L. y Knupp, P. Tetrahedral mesh improvement via optimization of the element condition number. *International Journal for Numerical Methods in Engineering*, 83: 1377–1391, 2002.

Freitag, L. y Plassmann, P. Local optimization-based simplicial mesh untangling and improvement. *International Journal for Numerical Methods in Engineering*, 49: 109–125, 2000.

Frey, P. y George, P. *Mesh Generation.* Hermes Sciense Publishing, Oxford, 2000.

Garratt, J. Observations in the nocturnal boundary layer. *Boundary-Layer Meteorology*, 22(1): 21–48, 1982.

Geai, P. Methode d'interpolation et de reconstitution tridimensionelle d'un champ de vent: le code d'analyse objetive MINERVE. Informe técnico, Electricitè de France, 1985.

George, P. y Borouchaki, H. *Delaunay Triangulation and Meshing: Aplication to Finite Elements.* Editions Hermes, París, 1998.

George, P., Hecht, F. y Saltel, E. Automatic mesh generator with specified boundary. *Computer Methods in Applied Mechanics and Engineering*, 92(3): 269–288, 1991. ISSN 0045-7825.

González-Yuste, J., Montenegro, R., Escobar, J., Montero, G. y Rodríguez, E. Implementation of a refinement/derefinement algorithm for tetrahedral meshes. En B. Topping y C. Soares, editores, *Proceedings of The Fourth International Conference on Engineering Computational Technology*, páginas 27–28. Civil-Comp Press, septiembre 2004a.

González-Yuste, J., Montenegro, R., Escobar, J., Montero, G. y Rodríguez, E. Local refinement of 3-D triangulations using object-oriented methods. *Advances in Engineering Software*, 35(10-11): 693–702, septiembre 2004b. ISSN 0965-9978.

González-Yuste, J., Rodríguez, E., Montenegro, R., Escobar, J. y Montero, G. Mesh adaptation with refinement/derefinement for a 3-d wind field model. En G.M. B.H.V. Topping y R. Montenegro, editores, *Proceedings of The Fifth International Conference on Engineering Computational Technology*. Civil-Comp Press, septiembre 2006. ISBN 1-905088-11-6.

Gross, S. y Reusken, A. Parallel multilevel tetrahedral grid refinement. *SIAM Journal on Scientific Computing*, 26(4): 1261–1288, 2005. ISSN 1064-8275.

Holland, J. *Adaption in Natural and Artificial Systems*. MIT Press, 1992.

Hrennikoff, A. Solution of problems in elasticity by the framework method. *Journal of Applied Mechanics*, 8: 169–175, 1941.

Kitada, T., Kaki, A., Ueda, H. y Peters, L. Estimation of vertical air motion from limited horizontal wind data - a numerical experiment. *Atmospheric Environment*, 17: 2181–2192, 1983.

Knupp, P.M. Achieving finite element mesh quality via optimization of the jacobian matrix norm and associated quantities. part ii-a frame work for volume mesh optimization and the condition number of the jacobian matrix. *International Journal for Numerical Methods in Engineering*, 48: 1165–1185, 2000.

Knupp, P.M. Algebraic mesh quality metrics. *SIAM Journal on Scientific Computing*, 23(1): 193–218, 2001. ISSN 1064-8275.

Lalas, D., Tombrou, M. y Petrakis, M. Comparison of the performance of some numerical wind energy siting codes in rough terrain. En *European Community Wind Energy Conference*. Herning, Denmark, 1988.

Levine, D. *A Parallel Genetic Algorithm for the Set Partitioning Problem*. Tesis Doctoral, Illinois Institute of Technology / Argonne National Laboratory, 1994.

Liu, A. y Joe, B. Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision. *Mathematics of Computation*, 65(215): 1183–1200, 1996.

Löhner, R. y Baum, J. Adaptive h-refinement on 3d unstructured grids for transient problems. *International Journal for Numerical Methods in Fluids*, 14: 1407–1419, junio 1992.

Löhner, R. y Parikh, P. Three-dimensional grid generation by advancing front method. *International Journal for Numerical Methods in Fluids*, 8: 1135–1149, 1998.

Montenegro, R., Montero, G., Escobar, J.M. y Rodríguez, E. Efficient strategies for adaptive 3-d mesh generation over complex orography. *Neural, Parallel & Scientific Computations*, 10(1): 57–76, 2002a. ISSN 1061-5369.

Montenegro, R., Montero, G., Escobar, J., Rodríguez, E. y González-Yuste, J. Tetrahedral mesh generation for environmental problems over complex terrains. *Lecture Notes in Computer Science*, 2329: 235–344, 2002b.

Montenegro, R., Montero, G., Escobar, J., Rodríguez, E. y González-Yuste, J. 3-d adaptative wind field simulation including effects of chimney emissions. En *Proceedings of The Sixth World Congress on Computational Mechanics (WCCM VI)*, septiembre 2004.

Montenegro, R., Montero, G., Escobar, J., Rodríguez, E. y González-Yuste, J. Wind field simulation using adaptive tetrahedral meshes. En *Innovation in Civil and Structural Engineering Computing*, capítulo 8, páginas 159–185. Saxe-Coburg Publications, 2005.

Montenegro, R., Plaza, A., Ferragut, L. y Asensio, M. Application of a nonlinear evolution model to fire propagation. *Nonlinear Analysis. Theory, Methods & Applications*, 30(5): 2873–2882, diciembre 1997.

Montero, G., Montenegro, R. y Escobar, J. A 3-d diagnostic model for wind field adjustment. *Journal of Wind Engineering and Industrial Aerodynamics*, 74–76: 249–261, abril 1998.

Montero, G., Rodríguez, E., Montenegro, R., Escobar, J.M. y González-Yuste, J.M. Genetic algorithms for an improved parameter estimation with local refinement of tetrahedral meshes in a wind model. *Advances in Engineering Software*, 36(1): 3–10, 2005. ISSN 0965-9978.

Montero, G. y Sanín, N. 3-d modelling of wind field adjustment using finite differences in a terrain conformal coordinate system. *Journal of Wind Engineering and Industrial Aerodynamics*, 89: 471–488, 2001.

Moussiopoulos, N., Flassak, T. y Knittel, G. A refined diagnostic wind model. *Environmental Software*, 3: 85–94, 1988.

Pain, C., Umpleby, A., de Oliveira, C. y Goddard, A. Tetrahedral mesh optimization for steady-state and transient finite element calculations. *Computer Methods in Applied Mechanics and Engineering*, 190: 3771–3796, 2001.

Palomino, I. y Martín, F. A simple method for spatial interpolation of th wind in complex terrain. *Journal of Applied Meteorology*, 34: 1678–1693, 1995.

Panofsky, H. y Dutton, J. *Atmospheric Turbulence*. John Wiley, New York, 1984.

Plaza, A. y Carey, G. Local refinement of simplicial grids based on the skeleton. *Applied Numerical Mathematics*, 32(2): 195–218, febrero 2000.

Ratto, C. *Modelling of Atmospheric Flow Fields*, capítulo The AIOLOS and WINDS Codes, páginas 421–431. World Scientific Publishing, Singapur, 1996a.

Ratto, C. An overview of mass-consistent models. En D. Lalas y C. Ratto, editores, *Modelling of Atmospheric Flow Fields*, páginas 379–400. World Scientific Publishing, Singapur, 1996b.

Rivara, M. A grid generator based on 4 triangles conforming mesh-refinement algorithms for triangulations. *International Journal for Numerical Methods in Engineering*, 24: 1343–1354, 1987.

Rivara, M. y Levin, C. A 3d refinement algorithm suitable for adaptive multigrid techniques. *Communications in Applied Numerical Methods*, 8: 281–290, 1992.

Rodríguez, E., Montero, G., Montenegro, R., Escobar, J. y González-Yuste, J. Parameter estimation in a three-dimensional wind field model using genetic algorithms. *Lecture Notes in Computer Science*, 2329: 950–959, 2002.

Rodríguez Barrera, E. *Modelización y simulación numérica de campos de viento mediante elementos finitos adaptativos en 3-D.* Tesis Doctoral, Universidad de Las Palmas de Gran Canaria, mayo 2004.

Ross, D., Smith, I., Manins, P. y Fox, D. Diagnostic wind field modelling for complex terrain: Model development and testing. *Journal of Applied Meteorology*, 27: 785–796, 1988.

Sherman, C. A mass-consistent model for wind fields over complex terrain. *Journal of Applied Meteorology*, 17: 312–319, 1978.

Stepanov, A. y Lee, M. The standard template library. Informe Técnico 95–11(R.1), HP Laboratories Technical Report, noviembre 1995. Revised version of A. Stepanov and M. Lee: The Standard Template Library, Technical Report X3J16/94-0095, WG21/N0482, ISO Programming Language C++ Project, May 1994.

Strang, G. y Fix, G. *An Analysis of the Finite Element Method.* Society for Industrial and Applied Mathematics, 1973.

Syswerda, G. Uniform crossover in genetic algorithms. En J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, páginas 2–9. San Mateo, CA: Morgan Kaufmann, 1989.

Tam, A., Ait-Ali-Yahia, D., Robichaud, M., Moore, A., Kozel, V. y Habashi, W. Anisotropic mesh adaptation for 3-d flows on structured and unstructured grids. *Computer Methods in Applied Mechanics and Engineering*, 189: 1205–1230, 2000.

Thompson, J., Soni, B. y Weatherill, N. *Handbook of Grid Generation*. CRC Press, Londres, 1999.

Tinsley Oden, J. Historical comments on finite elements. En *A history of scientific computing*, páginas 152–166. ACM Press, New York, NY, USA, 1990. ISBN 0-201-50814-1.

Tombrou, M. y Lalas, D. A telescoping procedure for local wind energy potential assessment. En W. Palz, editor, *European Community Wind Energy Conference*. H.S. Stephens & Associates, 1990. Madrid.

Whitley, D. GENITOR: A different genetic algorithm. En *Rocky Mountain Conference on Artificial Intelligence*, 1988.

Whitley, D. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. En *Third International Conference on Genetic Algorithms*, 1989.

Wilson, E. y Clough, R. Early finite element research at berkeley. En *Fifth U.S. National Conference on Computational Mechanics*, agosto 1999.

Winter, G., Montero, G., Ferragut, L. y Montenegro, R. Adaptive strategies using standard and mixed finite elements for wind field adjustment. *Solar Energy*, 54(1): 49–56, enero 1995.