UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
A R A G O N

SISTEMA GRAFICO DE GENERACION DE MALLAS PARA EL METODO DEL ELEMENTO FINITO.

T E S I S

QUE PARA OBTENER EL TITULO DE

INGENIERO CIVIL

P R E S E N T A

ADRIAN ALBERTO BICUÑA MALDONADO

SN. JUAN DE ARAGON, EDO. DE MEXICO 1992.

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

INDICE

INTRODUCCION Y OBJETIVO.

En la actualidad, el avance de la tecnologia ha permitido el desarrollo de modelos numéricos de fenómenos fisicos y de sistemas complejos de ingenieria que simulan el comportamiento real para diversas condiciones fisicas. Si la simulación del comportamiento se apega a la realidad, tales sistemas se pueden diseñar de manera confiable. También se pueden optimar funciones propias de cada sistema, como son el costo o el tiempo de operación, que están ligadas a los parámetros fisicos y restricciones utilizadas para el diseño del sistema.

El proceso de simulación comprende varios pasos de los cuales el modelado numérico es uno de ellos. De esta manera el proceso de simulación se inicia con la concepción del sistema ingenieril y las leyes y principios fisicos que gobiernan su comportamiento.

El paso siguiente consiste en desarrollar un modelo matemático que describa el funcionamiento del sistema bajo los principios preestablecidos. Este puede ser un paso dificil en el proceso de simulación, ya que es necesario estimar los efectos del medio en el cual el sistema debe funcionar, las caracteristicas de los materiales con que puede estar compuesto, seleccionar las leyes fisicas que predominen en él y transformar todo esto en un modelo matemático que describa eficientemente al sistema real.

Es deseable que el modelo matemático represente fielmente al

sistema real aunque a medida que esto se logre será difícil, o imposible, obtener información útil de éste por medio de procesos puramente matemáticos.

Una alternativa para la solución de modelos matemáticos complejos es recurrir a algún método numérico que proporcione una solución aproximada.

Con ésta alternativa, el modelo matemático establecido se discretiza, lo cual implica obtener un sistema de ecuaciones lineales simultaneas que caracterizan al modelo. La división adecuada de una región se obtiene con un número de partes cuya forma es tal que la proporción entre su ancho y largo es cercana a la unidad (forma regular). A éstas partes con forma regular se les conoce como elementos. Cada elemento esta unido con otro similar por medio de sus vértices llamados nodos. Se debe satisfacer así mismo, condiciones de continuidad entre las variables asociadas a cada nodo. De esta manera se obtiene un modelo discretizado.

El desarrollo del modelo discretizado es un paso crítico dentro del procedimiento de solución, pues involucra la selección y aplicación de algun método numérico para resolver el problema de manera eficiente y confiable.

El desarrollo de métodos numéricos avanzados tal como el método del elemento finito (MEF) constituyen una poderosa herramienta tanto para ingenieros de la práctica como para científicos que pretendan estudiar sistemas físicos en general. Por otro lado, el desarrollo de los nuevos computadores, que actualmente tienen una gran velocidad de procesamiento e importante tamaño de memoria, permiten que el MEF proporcione una gran flexibilidad para la solución de diversos problemas físicos, particularmente los de la rama de la mecánica de sólidos.

Algunos programas de elemento finito han sido diseñados de

tal manera que los datos de los elementos y nodos son introducidos secuencialmente con poca flexibilidad para hacer modificaciones en ellos una vez capturados.

Muchos problemas prácticos, consideran a miles de elementos y nodos, por consiguiente la tarea de preparar estos datos suele ser extremadamente larga y tediosa. Además, durante su preparación manual pueden cometerse fácilmente errores y permanecer sin ser detectados, produciendo resultados incorrectos.

Es importante eliminar al máximo ese tipo de errores. La mejor solución a este problema es crear y archivar los datos con la ayuda de un generador automático de mallas, en el cual los elementos, nodos y coordenadas, son generados automáticamente por la computadora utilizando un mínimo de datos de entrada para describir la geometría del sistema y el refinamiento deseado en la malla.

El problema de generar elementos finitos no es nuevo, ya que en los últimos veinte años se han desarrollado diferentes preprocesadores o generadores de mallas comerciales. Sin embargo estos se han diseñado para resolver problemas muy específicos y además no es posible hacer modificaciones en ellos, ya que no se dispone del código fuente y se desconoce el proceso específico del algoritmo empleado. Esto constituye una gran limitación para poder estudiar otro tipo de aplicaciones siendo prácticamente imposible implementar nuevas técnicas y procesos de malleo.

Para el MEF es necesario un almacenamiento dinámico, esto es, crear espacio de memoria conforme se necesite, ya que el número de datos por generar es variable, según el problema a estudiar. El lenguaje de programación *C* proporciona, entre otras, esta ventaja.

Recientemente el lenguaje *C* ha surgido como el seleccionado para desarrollar software profesional ya que ofrece: bloques de

líneas llamadas funciones, los programas son ejecutados con una mayor rapidez, un programa puede ser transportado de una máquina a otra con cambios pequeños o inclusive sin ninguno, el tamaño del programa generado es menor que el obtenido usando otro compilador.

Por todo lo anterior, en el departamento de Ingeniería Civil del Instituto de Investigaciones Eléctricas, se planteó la necesidad de desarrollar un generador de mallas que ayude a discretizar particularmente un modelo mecánico, en forma segura y rápida, para su utilización con el MEF. Este generador se implementó en lenguaje de programación C, por las ventajas descritas anteriormente. De esta manera se contará con el código fuente, conociendose con detalle el proceso que se utilizó para generar mallas de elementos finitos; asimismo se tendrá la posibilidad de modificarlo conforme surgan nuevas necesidades de aplicación para problemas específicos que se deseen resolver con ayuda del MEF.

FUNDAMENTOS BASICOS DEL METODO DEL ELEMENTO FINITO.

La mejor manera de resolver un problema fisico gobernado por una ecuación diferencial es obtener la solución analitica correspondiente. Sin embargo, hay muchas ocasiones en que esta solución es dificil, o imposible, de obtener. Cuando ésto se presenta la alternativa es recurrir a un método numérico que proporcione una solución aproximada.

Considerese una ecuación diferencial ordinaria en la cual no se puede encontrar alguna expresión analitica exacta para la solución y se recurre a construir una solución aproximada de la forma:

$$u(x) = c_i \phi_i(x) \quad (i=1,\ldots M) \tag{2.1}$$

donde $\phi_1(x),\ldots\phi_M(x)$ son un conjunto de M funciones linealmente independientes y $c_1,\ldots,c_M$ son las constantes que se deben determinar para la solución $u(x)$.

El primer paso en la solución es escoger el conjunto de funciones de aproximación $\phi_i(x)$. El siguiente paso es encontrar los coeficientes c_i que dan la mejor aproximación para el conjunto de funciones. De esta manera el problema de encontrar una función continua desconocida se transforma por el de encontrar un conjunto finito de valores que proporcione la mejor aproximación a la solución. Para encontrar este conjunto se requiere de algún

criterio. Hay varios métodos que proporcionan tal criterio y pueden clasificarse en tres grupos básicos:

- *El método variacional.*
- *El método de los pesos residuales.*
- *El método de las diferencias finitas.*

En lo que sigue, se hace mención a los dos primeros, el tercero puede consultarse en Burden (*Burden, 1985*). Para entender la formulación de estos métodos es necesario exponer algunos conceptos básicos.

2.1 Energía potencial.

La figura 2.1 muestra un cuerpo con peso P, sobre un plano horizontal sometido a una fuerza constante F, que produce un desplazamiento e. Si se supone nula la fricción, la fuerza ha efectuado un *trabajo* T = Fe (fuerza por desplazamiento).

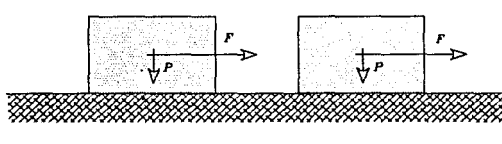


Fig.2.1 Desplazamiento de un cuerpo sobre una superficie sin fricción.

En el caso de la figura 2.1 el cuerpo no ha permanecido en reposo y el trabajo T es una *energía cinética*. Por otra parte, la fuerza F, produce desplazamientos internos de sus partículas, generando a su vez deformaciones en su forma.

Cuando se producen las defomaciones debidas a la fuerza F,

éstas han efectuado un trabajo T, que queda almacenado en el cuerpo y que por reversión, éste es capaz de devolver por ser elástico. Todo trabajo almacenado y que puede recuperarse, recibe el nombre de *energía potencial.*

2.2 Energía de deformación.

La figura 2.2 muestra una barra de material elástico sometida a una fuerza axial y gradual P, que produce un alargamiento D.

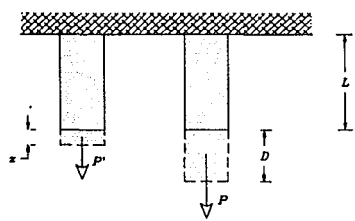


Fig.2.2 Deformación axial

Cuando la fuerza vale P' el alargamiento es z, entonces se tiene que:

$$D = \frac{PL}{AE} \qquad P = \frac{AED}{L}$$

$$z = \frac{P'L}{AE} \qquad P' = \frac{AEz}{L} z$$

Si P' es incrementada en dP'z, la deformación sufre un incremento dz y el trabajo un incremento igual a dT = (P' + dP')dz. Despreciando el producto dP' dz entonces dT = P'dz y el trabajo total efectuado por P durante la deformación D será:

$$U = \int_0^D P'zdz$$

sustituyendo P' en la integral:

$$U = \frac{AE}{L}\int_0^D z\,dz = \frac{AED^2}{2L}$$

de donde

$$U = \frac{AED^2}{2L} \qquad (2.2)$$

a esta última expresión se le conoce como la energía interna de deformación en una barra sometida a una fuerza axial gradual.

2.3 Método variacional.

Este método opera no en la ecuación diferencial, sino en un principio variacional, en el que la solución se asocia con el valor de equilibrio de una integral (funcional). Para las ecuaciones diferenciales derivadas de sistemas físicos, esta integral representa alguna forma de energía, de tal manera que si la solución es estable el valor de equilibrio es mínimo. Para comprender mejor este método es importante conocer el principio de la energía potencial total en el cual se basa.

La energía potencial total (Π) que se presenta en un sistema estructural es la suma de la energía potencial generada por las cargas aplicadas (Ω), más la energía de deformación interna (U) almacenada por la estructura:

$$\Pi = U + \Omega \qquad (2.3)$$

Considerese nuevamente la figura 2.2. las dos energías se pueden expresar en función de los desplazamientos, D, correspondientes del sistema. En este caso, la energía de deformación interna del sistema es

$$U = \frac{kD^2}{2} \qquad \text{donde } k = EA/L \qquad (2.4)$$

y la energía debida a la carga exterior es:

$$\Omega = - PD \qquad (2.5)$$

Concecuentemente, la energía potencial total del sistema esta dada por la ecuación:

$$\Pi = \frac{kD^2}{2} - PD \qquad (2.6)$$

Si el sistema está en equilibrio estable bajo la acción de la carga la energía potencial total del sistema debe tener un valor mínimo. Así, para pequeños desplazamientos de la barra en tracción, a partir de la posición de equilibrio, no debe haber cambios significativos en el valor de la energía potencial total.

Para que el sistema este en un punto de equilibrio es necesario que prevalesca la siguiente condición:

$$\frac{d\Pi}{dD} = 0 \qquad (2.7)$$

Además de cumplir con la expresión anterior, condición estacionaria, es necesario conocer si el punto es mínimo (estable). Para que un punto de equilibro sea un mínimo la segunda derivada de la expresión de la energía potencial total debe ser mayor que cero.

$$\frac{d^2\Pi}{dD^2} > 0 \qquad (2.8)$$

Con base en los conceptos expuestos en los párrafos anteriores se puede demostrar que el funcional asociado a la barra en tracción axial de la figura 2.2, es:

$$\Pi = U + \Omega = \frac{1}{2}\int_0^L c^2 EA\, dx - \int_0^L qu\, dx$$

$$= \int_0^L \left[\frac{1}{2}\left(\frac{du}{dx}\right)^2 EA - qu\right] dx \qquad (2.9)$$

donde $c = du/dx$.

El valor numérico de Π puede calcularse dando una ecuación

especifica a u = f(x). El cálculo de variaciones muestra que la ecuación particular u = g(x) la cual produce el valor numérico menor para Π, es la solución aproximada a la ecuación diferencial siguiente:

$$AE\frac{d^2u}{dx^2} + q = 0 \qquad (2.10)$$

y que es la que describe el comportamiento del problema.

Para aclarar mejor este método se recurre al ejemplo siguiente. Considerese la viga simplemente apoyada con una carga W repartida uniformemente como se aprecia en la figura 2.3

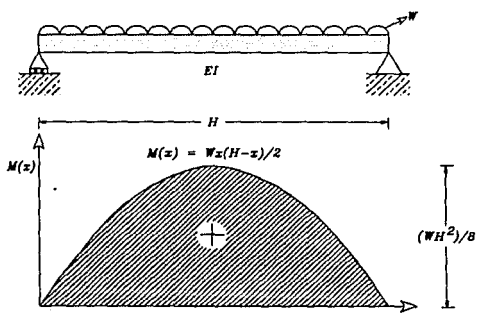


Fig.2.3 Viga simplemente apoyada con una carga uniformemente repartida y diagrama de momentos flexionantes.

La solución consiste en hallar una ecuación que se aproxime al desplazamiento para la viga simplemente apoyada. Supongase que la ecuación de prueba es del tipo senoidal:

$$y(x) = A\ sen\ \pi x/H \qquad (2.11)$$

Sustituyendo (2.11) en (2.9) y considerando que:

$$\frac{du}{dx} = \frac{dy}{dx} \; ; \quad q = M(x); \quad u = y(x)$$

se obtiene que:

$$\frac{dy}{dx} = \frac{A\pi}{H} \cos \frac{\pi x}{H} \tag{2.12}$$

el funcional asociado es:

$$\Pi = \int_0^H \left[\frac{EI}{2} \left[\frac{A\pi}{H} \cos\frac{\pi x}{H} \right]^2 + \left[\frac{Wx(H-x)}{2} \right] A \operatorname{sen}\frac{\pi x}{H} \right] dx \tag{2.13}$$

desarrollando y teniendo en cuenta algunas identidades trigonométricas, la integración de la ecuación (2.13) es:

$$\Pi = \frac{EI\pi^2 A^2}{4H} + \frac{2WAH^3}{\pi^3} \tag{2.14}$$

la ecuación (2.14) es la energía potencial total del sistema. Derivando esta ecuación e igualando a cero se tiene:

$$\frac{d\Pi}{dA} = \frac{EI\pi^2 A}{2H} + \frac{2WH^3}{\pi^3} = 0 \tag{2.15}$$

despejando en (2.15) a A

$$A = - \frac{4WH^2}{EI\pi^5} \tag{2.16}$$

sustituyendo (2.16) en (2.17) se obtiene entonces una solución aproximada:

$$y(x) = - \frac{4WH^4}{EI\pi^5} \operatorname{sen} \frac{\pi x}{H} \tag{2.17}$$

la segunda derivada debe cumplir con la condición (2.8) para que el sistema sea estable:

$$\frac{d^2y}{dx^2} = \frac{4WH^2}{EI\pi^3} \operatorname{sen} \frac{\pi x}{H} > 0$$

esta ultima expresión es positiva con lo cual la curvatura de la función corresponde a la de un valor mínimo y por lo tanto es

estable.

2.4 Métodos de los pesos residuales.

En estos métodos, una función de prueba se selecciona para aproximar la variable independiente y se substituye en la ecuación diferencial. Esta función generalmente no satisface a la ecuación. Como concecuencia, resulta un término de error o residuo **R(x)**. A este residuo, se le asocia con una ecuación de peso buscando el coeficiente que proporcione la " mejor " solución. Para encontrar este coeficiente se debe minimizar la integral del residuo y la función de peso sobre el sistema.

Supongase que la ecuación que se desea aproximar tiene la siguiente forma:

$$D\frac{d^2y}{dx^2} + Q = 0 \qquad (2.18)$$

con las condiciones de frontera $y(0) = y_0$ y $y(H) = y_H$ sustituyendo en (2.22) la función $y = h(x)$ se obtiene:

$$D\frac{d^2h(x)}{dx^2} + Q = R(x) \neq 0 \qquad (2.19)$$

donde la función $y(h)$ no satisface la ecuación (2.18). El método de los pesos residuales requiere que:

$$\int_0^H W_i(x)R(x)dx = 0 \qquad (2.20)$$

El residuo **R(x)** se multiplica por una función de peso $W_i(x)$, además se requiere que la integral del producto sea cero. El número de funciones de peso es igual al número de coeficientes desconocidos en la solución aproximada. Las funciones de peso pueden ser escogidas de varias formas y cada una corresponde a un método diferente.

Entre los diferentes métodos de los pesos residuales se pueden mencionar:

- *El método de colocación.*
- *El método del subdominio.*
- *El método de Galerkin.*
- *El método de los mínimos cuadrados.*

Dentro de los métodos de los pesos residuales el más empleado es el de *Galerkin* ya que proporciona ecuaciones que son idénticas a las obtenidas por medio de principios variacionales, y que se describe brevemente en el inciso siguiente. Los métodos restantes pueden consultarse en Segerling (*Segerling, 1984*).

2.5 El método de Galerkin.

Este método usa la misma función para $W_i(x)$ que fué utilizada en la ecuación de aproximación. Para ilustrar este método se presenta un ejemplo de aplicación.

Considerese nuevamente, la viga simplemente soportada con una carga uniformemente repartida, figura 2.3, representada por la ecuación

$$EI \frac{d^2y}{dx^2} + M(x) = 0 \qquad (2.21)$$

que describe la deflexión de la misma y con las condiciones de frontera $y(0) = 0$, $y(H) = 0$.

El coeficiente EI representa la resistencia de la viga a la flexión y M(x) es la ecuación del momento flexionante dada por:

$$M(x) = \frac{Wx(H - x)}{2} \qquad (2.22)$$

La ecuación (2.11) cumple con las condiciones de frontera. Asimismo, esta ecuación se escoge como función de peso:

$$W_1(x) = \text{sen } \frac{\pi x}{H} \qquad (2.23)$$

Cuando se usa el método de Galerkin, la ecuación (2.20) es calculada usando la misma función $W_1(x)$ que fué utilizada para obtener el residuo R(x). En este ejemplo hay sólo una función de peso y para calcular la ecuación residuo se deriva la ecuación (2.11):

$$\frac{d^2y}{dx^2} = \frac{A\pi^2}{H^2} \text{ sen } \frac{\pi x}{H} \qquad (2.24)$$

sustituyendo (2.22) y (2.11) en (2.21) se obtiene el residuo R(x).

$$R(x) = - EI\frac{A\pi^2}{H^2} \text{ sen } \frac{\pi x}{H} - M(x) \qquad (2.25)$$

Con la ecuación (2.25) se sustituye a ésta y a (2.23) en $\int W_1(x)R(x)dx$, ecuación (2.20), y se calcula la integral (2.26).

$$\int_0^H \text{sen } \frac{\pi x}{H} \left[- EI\frac{A\pi^2}{H^2} \text{ sen } \frac{\pi x}{H} - \frac{Wx(H - x)}{2} \right] dx = 0 \qquad (2.26)$$

Integrando (2.26) se obtiene:

$$- \frac{EIA\pi^2}{2H} - \frac{2WH^3}{\pi^3} = 0 \qquad (2.27)$$

despejando a A en (2.27)

$$A = - \frac{4WH^4}{EI\pi^5} \qquad (2.28)$$

sustituyendo a (2.28) en (2.27) la solución aproximada es:

$$y(x) = -\frac{4WH^4W}{EI\pi^8}\ \text{sen}\ \frac{\pi x}{H} \qquad (2.29)$$

que resulta la misma que utilizando el método variacional. El método de Galerkin es la base fundamental del MEF.

2.6 Generalidades del Método del Elemento Finito.

Actualmente existen técnicas aproximadas para estudiar el comportamiento de estructuras continuas. El MEF permite estudiar dichas estructuras transformando el problema continuo a otro de forma discretizada. Este método es una de las técnicas de uso común dada su flexibilidad y facilidad de adaptación en sistemas de cómputo. Las computadoras digitales han facilitado el estudio de grandes problemas de análisis de estructuras lo cual ha permitido resolver problemas de gran complejidad.

Las estructuras delgadas tales como placas y láminas curvas, generalmente se discretizan en una malla de elementos triangulares o cuadrangulares cubriendo toda la superficie media; en las estructuras sólidas se pueden utilizar elementos tetraédricos o hexaédricos para formar el volumen. Los elementos de la malla se denominan *elementos finitos*.

Cuando un elemento estructural tiene curvatura, lo usual es disminuir el tamaño del elemento con el fin de hacer despreciable el efecto de la curvatura.

La selección de los puntos nodales en una estructura dada es muy importante. Generalmente los nodos se localizan con el propósito de crear zonas de uniformidad en el volumen de material. Asimismo, las particularidades de carga, tales como los puntos de carga y de apoyo, deben escogerse, en lo posible, como nodos.

La *densidad de los nodos*, es decir, el número de nodos por unidad de volumen del material, es una función de la rapidez de cambio de la geometría del material y de la carga con el espacio. Un cambio rápido implica una densidad nodal alta.

Cuando se aplica el MEF en la solución de problemas de mecánica de estructuras se desarrollan las siguientes etapas:

1.- Discretizar la región. Dividir la estructura continua en elementos finitos; para ello, es conveniente emplear un programa generador de mallas, llamado preprocesador. Esto incluye la localización y numeración de los puntos nodales, así como la definición de propiedades geométricas y mecánicas de los elementos generados.

2.- Formular las propiedades de cada elemento. En análisis estructural, esto significa determinar las cargas nodales asociadas con los estados de deformación que son permitidos en los elementos.

3.- Ensamblar los elementos finitos para formar el modelo idealizado de la estructura.

4.- Aplicar las cargas conocidas, fuerzas nodales y/o momentos para el análisis de esfuerzos.

5.- Especificar como esta apoyada la estructura, este paso involucra a varias series de valores de desplazamientos nodales conocidos (los cuales frecuentemente son cero).

6.- Resolver el sistema de ecuaciones algebraicas simultáneas, determinando así los valores de los grados de libertad que permiten calcular las deformaciones y esfuerzos.

7.- Resultados finales. Calcular la deformación de cada

elemento con la ayuda de los valores de los grados de libertad en los nodos y calcular los campos de deformación por medio de la interpolación. Finalmente calcular los esfuerzos a partir de los valores de la deformación.

La primera etapa de este proceso es la que aquí se desarrolla principalmente.

CAPITULO 3

EL PROBLEMA DE GENERACION DE MALLAS.

Como se mencionó en la introducción de este trabajo, la generación automática de mallas consiste en buscar un método que nos ayude a generar los elementos finitos no sólo rápidamente sino que tengan características tales que proporcionen una exactitud adecuada de acuerdo al problema en estudio.

Con base en la experiencia, se sabe que para lograr una mejor exactitud es necesario una concentración mayor de elementos en donde se espere que la variación del valor independiente a evaluar sea importante. A esta concentración de elementos se le conoce como refinamiento.

Aunque el refinamiento de una región sea deseable para evitar inestabilidad en los cálculos, el costo y tiempo computacional, principalmente al resolver el sistema de ecuaciones, puede crecer tanto que sea inadmisible utilizar la malla generada. Además, un refinamiento con elementos finitos muy distorsionados puede provocar inestabilidad en los cálculos. En la práctica es más conveniente tener pocos elementos de forma regular, que muchos de forma irregular. Se entiende como regular a la relación entre el ancho y largo del elemento; a esta relación se le denomina radio del elemento. Dicho radio deberá ser, en lo posible, muy cercano a la unidad.

Asi, la forma de un elemento es aceptable a medida que sea compacta y regular; será inaceptable cuando su radio se aleje de

la unidad, cuando los ángulos de sus esquinas sean marcadamente diferentes uno de otro cuando sus lados son curveados y si los nodos intermedios de un borde estan espaciados de manera no uniforme.

Cada elemento tiene sensibilidad diferente a la distorsión de su geometria. Por esta razón, para generar una malla de un modelo discreto aceptable, hay que tomar en cuenta las siguientes condiciones: mantener los radios cercanos a la unidad, que los ángulos de las esquinas de elementos cuadriláteros sean cercanos a 90• y para elementos triangulares cercanos a 60•, mantener los nodos intermedios a la mitad del lado donde se encuentran y conservar rectos los lados Sin embargo, los elementos con un radio grande pueden ser usados en regiones donde los cambios en el valor de la variable independiente sean casi cero.

No se puede considerar que un elemento sea mejor que otro. Un elemento que trabaja bien en una situación puede trabajar mal en otra. La experiencia del analista definirá como se comportan cierto tipo de elementos en cada problema.

Los elementos triangulares y rectangulares pueden ser usados en regiones regulares Sin embargo, el elemento triangular se adapta mejor a las regiones irregulares

Una malla con todos los elementos de un mismo tamaño y forma no es necesaria, sin embargo, se debe hacer una transición adecuada entre tamaños ya que una transición brusca se traducirá en una pérdida de exactitud. La habilidad para variar el tamaño del elemento es una ventaja importante que tiene el elemento triangular.

A continuación se presenta un estudio bibliográfico y los algoritmos seleccionados, que sirvieron de base para definir las caracteristicas del preprocesador o generador de mallas para el

MEF, desarrollado en este trabajo.

3.1 Estudio bibliográfico de algoritmos generadores de mallas.

Este estudio pretende dar ha conocer la amplia variedad de técnicas, que se han desarrollado en las dos últimas décadas, sobre generación de mallas. Para ello se seleccionó un cierto número de articulos clasificandolos según la técnica, época y su aportación dentro de su género para discretizar una región.

Uno de los motivos iniciales para la investigación en la generación de mallas fue reducir el tiempo invertido en la creación de los elementos finitos, los cuales pueden requerir hasta la mitad del tiempo que se emplea en el análisis del MEF.

En la década de los setentas la atención fue centrada para generar mallas dentro de regiones con forma irregular, principalmente en fronteras curveadas. También se comenzó a investigar en el desarrollo de métodos de adaptación de mallas consistentes en el refinamiento local y/o global de la malla mejorando la exactitud de los cálculos.

Posteriormente, se hicieron adaptaciones que incluían las caracteristicas de optimar y refinar la malla, mostrando ser de importancia cuando se modelaban regiones en las que actuaban cargas concentradas o esfuerzos elevados.

Cuando la generación de mallas comenzó a tener bases más firmes, muchos métodos y algoritmos diferentes se desarrollaron para resolver casos particulares. Conforme las computadoras comenzaron a ser aplicadas ampliamente, se automatizó más el proceso de generación de mallas, sin embargo aún fue necesaria la intervención del usuario en varios pasos del proceso.

Dentro de los trabajos realizados en esta época se pueden mencionar el de Zienkiewicz, quien utiliza la tranformación de coordenadas con elementos isoparamétricos, el de Gordon, quien introduce el uso de funciones de interpolación mezcladas y posteriormente propuso un mapeo llamado transfinito. También se puede mencionar el trabajo de Cook, quien introduce las coordenadas naturales en el proceso de generación, además consideró la generación de mallas en tres dimensiones, Durocher introduce la automatización en la reducción del ancho de banda. El lector puede recurrir al trabajo de Tracker quien cita ochenta artículos de esta década sobre el tema.

La década de los ochentas se caracterizó por el desarrollo de generadores automáticos, en los que se trato de integrar muchos de los procesos que implica mallar una región irregular en la computadora. La automatización del proceso ha sido casi total reduciendo la intervención del usuario. Así, surgen los métodos como el de la *triangulación automática* y el del *quadtree* modificado. En ellos no es necesario descomponer una zona compleja en varias simples y buscar una relación entre ellas para asegurar su continuidad, como era necesario en los métodos hasta el momento conocidos. Ahora el usuario sólo necesita especificar la información acerca de la geometría de la frontera y del tamaño del elemento seleccionado a lo largo de esta frontera (Deljouie-Rakhshandeh, 1990).

Al respecto ha surgido una basta literatura, ver Ho-le quien cita y clasifica varios artículos recientes sobre el tema. Los dos métodos señalados son los que más aplicaciones han tenido, y entre estos han surgido nuevas investigaciones que los mejoran o combinan. En lo que sigue se hará mención a los métodos de malleo con triángulos y cuadriláteros.

3.2 Triangulación automática

Estos métodos de generación de mallas emplean procesos geométricos para generar los puntos nodales en el interior de la región para después buscar una relación que los una.

Las ventajas de la triangulación automática son:

a) No esta limitado a las fronteras del dominio, y la orientación de este dominio no influye en la malla final.

b) Claros o agujeros dentro del dominio pueden ser abordados fácilmente, la subdivisión del dominio en poligonos convexos no es necesaria.

c) No se necesita establecer alguna relación matemática entre las fronteras.

3.2.1 Generación de nodos interiores.

Las primeras técnicas generaban los nodos interiores sobreponiendo una malla burda auxiliar de elementos rectangulares, consultar Shaw, que cubria la región en su totalidad. En la figura 3.1 el ancho y alto de los rectángulos es de 2 y $\sqrt{3}$, respectivamente, y son colocados de manera desfasada, lineas discontinuas. A continuación se calcula el centro de gravedad de los rectángulos insertando en cada uno un nodo interior, el sistema total de nodos forma triángulos equiláteros, figura 3.1 lineas continuas, sin embargo la forma de los elementos triangulares a lo largo de la frontera se controla por las caracteristicas de ésta.

El problema que resta por resolver es como distribuir los

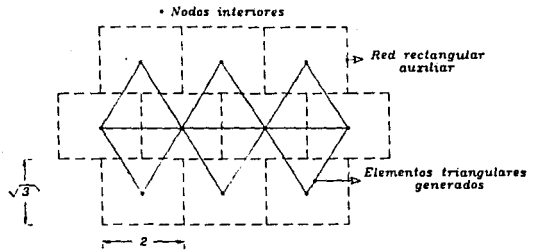


Fig.3.1 Una red rectangular auxiliar para generar nodos interiores.

rectángulos auxiliares en la frontera de la región, tal que su número sea adecuado para generar los elementos triangulares. Aunque posteriormente se hicieron modificaciones estas no resultaron satisfactorias.

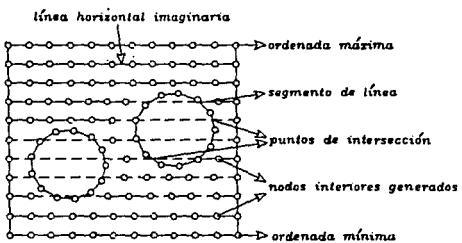


Fig.3.1a Se generan nodos interiores con líneas horizontales.

Posteriormente se desarrolló un método simple para generar nodos interiores en cualquier región (S.H.Lo, 1985). En este método se determinan las ordenadas máxima y mínima de la región por discretizar, en donde se dibujan lineas horizontales imaginarias con una distancia entre ellas igual al tamaño

promedio de los elementos deseados. Se determinan los puntos de intersección entre las lineas horizontales imaginarias y la región, ver figura 3.1a. En el dominio se dibuja la línea horizontal imaginaria que tendrá un número par de puntos de intersección. Con estas intersecciones son creados segmentos de linea donde los nodos pueden ser generados dependiendo de si estan cerca de la frontera y/o alejados de los nodos ya generados. Después de terminar con este segmento se continúa con los demás y con las lineas restantes.

3.2.2 Triangulación de Delaunay.

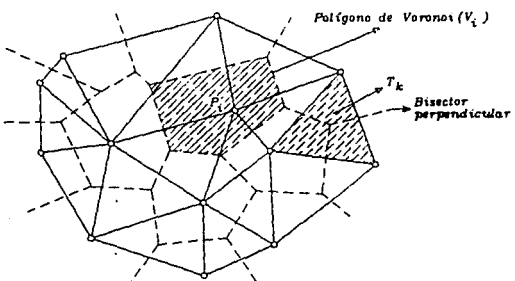


Fig.3.2 Mosaico de Dirichlet (líneas discontinuas) y triangulación de Delaunay (líneas continuas).

La triangulación de Delaunay es usualmente definida en términos de un diagrama auxiliar llamado polígono de Voronoi. Considerese el caso en dos dimensiones, figura 3.2. Sean p_1, p_2,...., p_N puntos nodales distintos en el plano $\mathbf{R}^2$, y las series V_i, $1 \leq i \leq N$, donde V_i representa una región, cuyos puntos nodales son los más cercanos al nodo p_i que a otro. Así, V_i es un polígono convexo, usualmente llamado polígono de Voronoi, cuyas fronteras son segmentos de los bisectores perpendiculares a las lineas que unen los nodos p_i y p_j cuando V_i y V_j son contiguas.

El conjunto de polígonos de Voronoi $\{V_i\}$ $i = 1, N$ es llamado mosaico de Dirichlet

En general, cada vértice de un polígono de Voronoi es compartido por tres polígonos vecinos tal que conectando los tres puntos nodales (p_i) de estos formarán un triángulo representado por T_k. La serie de triángulos $\{T_k\}$ es llamada *la triangulación de Delaunay*.

Una propiedad importante de la triangulación de Delaunay es que tres puntos nodales formarán un triángulo de Delaunay si y sólo si el circumcírculo definido por estos tres nodos no contiene otro punto nodal en su interior.

En el proceso de triangulación de Delaunay se utiliza, comunmente, la siguiente terminología. Tres puntos nodales no colineales definen a un triángulo y a un círculo llamado el circumcírculo del triángulo. El área del circumcírculo es llamado el circumdisco mientras que el radio y el centro son el circumradio y el circumcentro respectivamente.

Otra propiedad importante de la triangulación de Delaunay es que está es adecuada para generar elementos finitos triangulares cuya forma es muy próxima al triángulo equilátero, y por ello su importancia en el proceso de generación de mallas con triángulos.

En el algoritmo, tres puntos nodales dados formarán un triángulo de Delaunay si y sólo si el circumdisco definido por estos nodos no contiene ningún otro nodo en su interior, esta propiedad se ilustra en la figura 3.3.

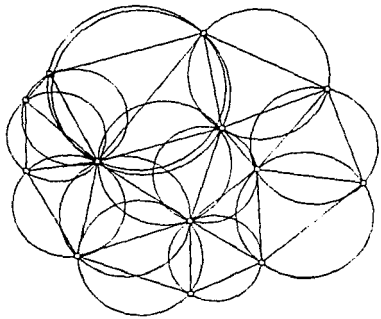


Fig 3.3 Ningún punto nodal se encuentra dentro de algún circumcírculo.

Lo expuesto en los párrafos anteriores es tan sólo la definición del método y sus propiedades. A continuación se presenta el proceso que sigue el algoritmo.

Se escoge la región y se definen los nodos de su frontera dependiendo del número de estos, se coloca en la región algunos nodos iniciales tal que formen triángulos, en la figura 3.4 se colocaron cuatro nodos inicialmente. También, si la región es simple, el algoritmo se puede inicializar por el cálculo de las coordenadas de tres puntos que formen un triángulo inicial T_0 que circunde los puntos nodales que se insertarán. El algoritmo opera manteniendo una lista de tripletes de puntos nodales los cuales representan los triángulos de Delaunay. Además, asociadas con cada triángulo se encuentran las coordenadas x y y del circumcentro y del circunradio. Estas coordenadas se emplean para identificar los triángulos cuyos circumdiscos contengan al nuevo punto nodal por insertar, ver figura 3.4.

Para cada circumdisco donde se encontró al punto nodal el lado del triángulo asociado a este es removido. Como se muestra en la figura 3.5.

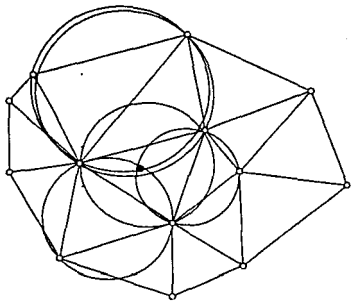


Fig.3.4 Intersección de un punto nuevo.

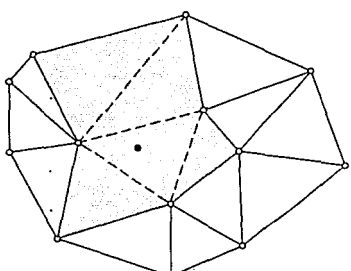


Fig.3.5 Polígono de inserción (región sombreada).

Con la unión de los triángulos de lados que se removieron se forma lo que se llama el polígono de inserción, zona sombreada figura 3.5, que contiene al nuevo punto nodal. Se puede ver que la unión entre el nodo insertado, contenido en el polígono de inserción, y los nodos fronterizos de éste pueden realizarse por medio de líneas rectas. Esto es, una nueva triangulación de la región es formada que cumple con las propiedades de Delaunay, ver figura 3.6.

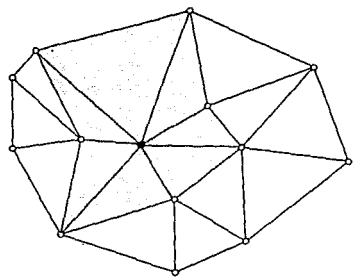


Fig.3.6 Triangulación local (zona sombreada).

El uso repetitivo del algoritmo de inserción permite que todos los puntos nodales se adicionen.

3.3 Mallas basadas en aproximaciones.

El método del *quadtree* nos permite tener una representación conveniente de objetos en dos dimensiones en un árbol que puede ser manipulado eficientemente utilizando simples operaciones.

En la técnica del *quadtree*, el objeto de interés es colocado dentro de un cuadrado que tiene definido un sistema coordenado por medio del cual se identifica a los cuadrantes empleados en la representación del objeto.

El cuadrado es dividido en cuatro cuadrantes y se comprueba el estado de cada uno, si permanecen dentro del objeto (lleno) o fuera del objeto (vacio), o si sólo parte de éste se encuentra dentro del objeto (parcial).

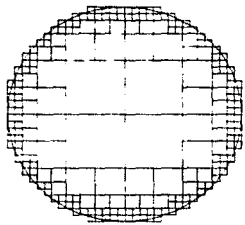


Fig 3.7 Representación de un círculo por quadtree

Los cuadrantes que están llenos o vacios son almacenados mientras que los cuadrantes parcialmente llenos son nuevamente divididos en cuatro subcuadrantes. Estos subcuadrantes son examinados de la misma manera que los cuadrantes iniciales, y el proceso continúa hasta llegar al nivel de resolución deseado.

Como se puede apreciar en la figura 3.7, la representación de un circulo por el método del *quadtree* es una discretización por si misma. Sin embargo, si se revisan los requerimientos y restricciones de los elementos de malla, se comprenderá el porque esta aproximación es imprácticа: primero, la exactitud de la solución del MEF depende de la distribución de los elementos de la malla, en la figura 3.7 los elementos estan distribuidos en mayor número en las fronteras induciendo cambios grandes de la variable a evaluar donde en realidad no los hay, y segundo el costo del análisis es alto en razón que el número de elementos en la malla puede ser muy grande, dependiendo de la forma de las fronteras y de la aproximación que se desee del modelo real.

El primer problema puede ser resuelto simplemente por el requerimiento de que todos los cuadrantes sean subdivididos en un nivel mínimo. El otro problema puede solucionarse modificando el código original como acontinuación se expone.

Si se eliminan las esquinas indeseables de 90• causadas por las fronteras no verticales y no horizontales. Se obtendrá una

representación geométrica mejorada, de los objetos, con un nivel de resolución bajo. Esto se logra empleando cuadrantes que puedan tener esquinas cortadas mejorando los ángulos que se pueden emplear. Con ésta situación el número de ángulos disponibles se incrementa, permitiendo mejorar la aproximación para un nivel dado.

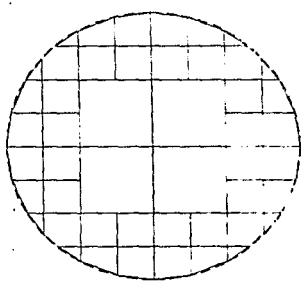


Fig.3.8 Representación de un círculo por quadtree modificado

Con los cambios anteriores el método del *quadtree* modificado representa la estructura básica que define una malla de elementos finitos válida, ver figura 3.8.

En el método del *quadtree* para repesentar el estado vacio, parcial y lleno de los subcuadrantes se emplean los números cero uno y dos respectivamente. Pero además de esto, la aproximación del *quadtree* modificado emplea el número tres para representar al subcuadrante cortado.

Cuando el algoritmo encuentra un subcuadrante cortado toda la información adicional requerida es recuperada en un arreglo separado que almacena los dos puntos finales de la diagonal del segmento de linea que corta al subcuadrante. En la figura 3.8 se muestra la representación del circulo por el método del *quadtree*

modificado, comparando esta figura con la 3.7 se observa que para el mismo circulo el método del *quadtree* modificado produce una representación más aproximada con un almacenamiento de datos menor. Este es el mismo caso para muchas geometrías y es verdad para todas las fronteras que contienen curvas o ángulos pronunciados.

Una vez que se tiene el modelo aproximado del *quadtree* modificado se adiciona a cada subcuadrante una diagonal para generar elementos triangulares, adicionalmente se mejora la malla empleando algún proceso de suavizado

Por si misma, la aproximación del *quadtree* modificado con la diagonal simple no produce mallas de elementos satisfactorios. Es necesario realizar implementaciones adicionales: crear una transición adecuada entre cuadrantes de diferentes niveles, asegurar que los elementos tengan radios razonables, y obtener la mejor aproximación geométrica posible. Para más información al respecto consultar el articulo de Yerry (*Yerry, 1983*).

3.4 Mapeo de puntos

Las primeras técnicas de malleo fueron basadas en métodos de mapeo (Zienkiewicz, 1971). Este proceso parte de la idea que es más sencillo discretizar una región de forma simple y regular. La región original se relaciona por medio de una serie de funciones que la transforma a una forma tal que sea fácil de mallar. Los nodos se generan en esta región simple se transfieren a la región original mediante la inversa de la función de transformación utilizada.

Los métodos de mapeo han sido la metodologia de varios programas comerciales que generan mallas. Una de las ventajas de este método es que se generan elementos cuadrangulares

facilmente. La desventaja es que un dominio complejo debe de ser subdividido manualmente por el usuario en varias regiones simples para aplicar el procedimiento. Otra limitación es el grado de automatización que se puede lograr ya que es necesario buscar una relación entre las subregiones vecinas en que ha sido dividida la región, y también hay un número limitado de geometrías que se pueden mallar satisfactoriamente.

Zienkiewicz y Phillips utilizaron un mapeo isoparamétrico curvilíneo de cuadriláteros, el cual relaciona los sistemas curvilíneal y cartesiano de coordenadas. Considérese el caso paticular de un cuadrilátero parabólico (es decir, que tiene lados curvos) como el de la figura 3.9 en el cual se conocen las coordenadas x y y de ocho nodos. Por medio de las relaciónes siguientes se mapea un punto cualquiera dentro de la región del cuadrilátero:

$$x = \sum_{i=1}^{8} N_i x_i$$

$$y = \sum_{i=1}^{8} N_i y_i \qquad (3.1)$$

$$z = \sum_{i=1}^{8} N_i z_i$$

en donde las N_i son funciones de forma o de interpolación

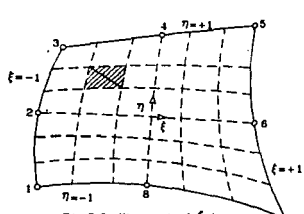


Fig.3.9 Elemento básico

definidas en términos de un sistema coordenado curvilíneo ξ y η; estas variables estan acotadas entre 1 y -1 como se observa en la figura 3.9. Las cuales pueden ser diferentes para cada tipo de elemento. Por ejemplo, para el elemento de la figura 3.9, sus funciones de forma son:

$$N_1 = -\frac{1}{4(1-\xi)\ (1-\eta)\ (\xi+\eta+1)} \qquad (3.2)$$

$$N_2 = \frac{1}{2(1-\xi)\ (1-\eta^2)}\ , \text{ etc}$$

dado que se utiliza una interpolación cuadrática en cada lado.

Si se conocen las coordenadas de los puntos nodales, entonces las coordenadas en el sistema cartesiano de cualquier punto especifico en ξ y η se pueden calcular con ayuda de las ecuaciones (3.1).

Apartir de este proceso se puede generar automáticamente una malla con cualquier refinamiento y emplear funciones de forma de cualquier grado, cuadráticas o cúbicas. Durocher y Gasper utilizaron el mapeo de Zienkiewicz y Phillips en un programa de computadora (Durocher, 1979), generando nodos, coordenadas y además la información necesaria de interconexión de los elementos en dos dimensiones.

El usuario define las subregiones y posteriormente se busca una relación entre estas con un elemento que puede ser un triángulo de lados rectos con tres o seis nodos, o con un triángulo isoparamétrico de seis nodos, o con un cuadrilátero isoparamétrico con cuatro u ocho nodos.

Otro concepto importante que utilizó Zienkiewicz es el diagrama llave. En este concepto la supeficie de la región es descompuesta en una serie de subregiones. Las cuales son

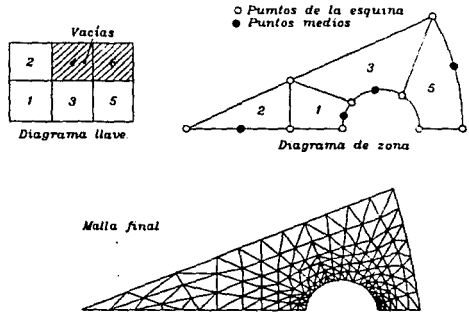


Fig 3.10 Transformación de coordenadas

relacionadas por medio de rectángulos que son ensamblados de la misma forma como lo estaban en la región original, ver figura 3.10. Las zonas vacías son adicionadas estableciendo rectángulos vacios en el diagrama llave.

El concepto de diagrama llave se lleva a cabo estableciendo un sistema de coordenadas naturales que es usado para generalizar los elementos finitos. La malla requerida es generada en el rectángulo y la malla verdadera en la subregión es obtenida usando las ecuaciones (3.2).

Con base en las técnicas de malleo presentadas hasta aquí, se han desarrollado un sin número de variantes que sería difícil presentarlas en este trabajo. Sin embargo, se han presentado los conceptos básicos que rigen este tipo de técnicas, y sobre los cuales se seleccionaron los algoritmos que se presentan en lo que sigue.

3.5 Descripción del algoritmo generador de elementos cuadrangulares.

A partir del estudio bibliográfico que se realizó fueron seleccionados dos artículos por ser los más recientes y que presentan técnicas nuevas de malleo. Uno de ellos genera elementos cuadrangulares (Cheng, 1989) y el otro elementos triangulares (Deljoui-Rakhshandeh, 1990). Las técnicas que se exponen en estos dos artículos han sido implementadas en lenguaje C. A continuación se describe el proceso para generar mallas con elementos finitos cuadrangulares.

Este algoritmo emplea el " método de la etiqueta asignada ", mediante el cual se proporciona la información necesaria para generar los elementos finitos. La malla se genera en forma paralela en todas las subregiones individuales. No es necesario comprobar la conformidad de los elementos entre las subregiones ya que ésta se asegura en forma automática. El algoritmo permite un refinamiento global o selectivo de la región. La geometría del objeto y el refinamiento deseado son los únicos datos que se introducen.

3.5.1 Definiciones.

Una red es una superficie con una serie de puntos, llamados nodos, y segmentos de líneas rectas. Tal que cada nodo es un extremo de un segmento además dos segmentos se intersectan sólo en sus extremos. También se debe tomar en cuenta que una red divide a la región en subregiones cada una limitada por segmentos de líneas rectas.

Un polígono es convexo cuando todos sus ángulos interiores son menores o iguales a 180° . Si todas las subregiones límites de una red son cuadriláteros convexos entonces esta es referida como una

red cuadrilátera. Adicionalmente una red es llamada cuadrilátera regular si la forma de sus subregiones no es muy larga o muy angosta. Dos subregiones **sb1** y **sb2** se dice estar adyacentes una de otra si comparten un borde común.

Sea **F**, el conjunto de todas las subregiones de una red cuadrilátera regular **P**. Un *nivel de subdivisión asignado* **S**, de la red **P** es una función definida en las subregiones **F**, tal que $S:F \to N \cup \{0\}$ donde **N** es el conjunto de todos los enteros positivos. **S(sb)** es llamado el nivel de subdivisión de la subregión **sb**, donde $sb \in F$. Dada una red cuadrilátera regular P, una red cuadrilátera P^* se le nombra una *malla subdividida* de P si cada elemento de P^* es una subserie de una subregión de P.

Las bases de partida para desarrollar este algoritmo pueden formularse de la siguiente manera. Dada una red cuadrilátera regular P y un nivel de subdivisión asignado por S en P, se desea generar una malla subdividida P^* de P tal que tenga las siguientes características: cada subregión **sb** de P sea subdividida en un mínimo de $4^{s(sb)}$ elementos, la forma de los elementos generados en P^* sea regular y la malla subdividida resultante P^* pueda sufrir modificaciones locales sin afectar el tamaño o forma de alguno de los elementos restantes. Las características listadas son el objetivo de este algoritmo.

Para cumplir con las características del párrafo anterior el algoritmo se apoya en los siguientes procedimientos:

- Método de la etiqueta asignada.
- Asignación de una etiqueta admisible.
- Subdivisión balanceada.
- Subdivisión no balanceada.

estos conceptos se definen a continuación.

3.5.2 Método de la etiqueta asignada.

Sea P una red cuadrilátera regular, con V y F como las series de vértices y subregiones de P respectivamente. Además considerar a un asignador de niveles de subdivisión S en P. Un asignador de etiquetas al vértice, L, de P con respecto a S es una función que $L: V \to N \cup \{0\}$, tal que $L(v) = \max \{S(sb) | sb \in F \text{ y } v \text{ es un vértice de } sb\}$. A L(v) se le llama la etiqueta de v con respecto a L. Un vértice v es un *vértice soportado* de L si L(v) ≠ 0, la figura 3.11 resume este proceso.

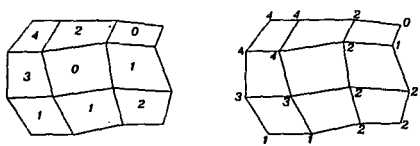


Fig.3.11 Ejemplo de nivel de subdivisión asignado y asignamiento de etiqueta al vértice.

3.5.3 Asignación de una etiqueta admisible.

El algoritmo genera la malla subdividida, P^*, de P comenzando con un tipo de asignamiento especial de etiqueta al vértice, llamado *asignación de una etiqueta admisible*. El cual se define en el párrafo siguiente.

Sean L y G los asignadores de las etiquetas a los vértices de P. A G se le conoce como una *extensión* de L si los valores de G son, por lo menos, iguales a los vértices de soporte (etiquetas no cero) de L, por ejemplo G(v) = L(v) si L(v) > 0. Por lo tanto, si G es una extensión de L entonces G(v) ≥ L(v) para cada vértice v de P. Así a G se le denomina una extensión de etiquetas admisibles con respecto a L, si la siguiente condición se cumple para todas

las subregiones **sb** de P: *si las etiquetas de dos vértices adyacentes de* **sb** *en* **G** *son no cero entonces por lo menos una de las dos etiquetas de los vértices restantes deben de ser no cero.*

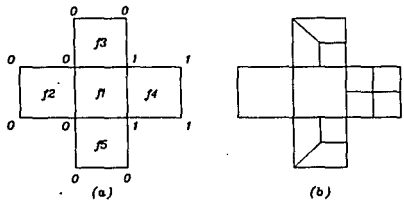


Fig 3.12 Subdivisión inadmisible.

Como el proceso de subdivisión se maneja unicamente por las etiquetas de los vértices, la construcción de la malla subdividida, P^*, utiliza la asignación de etiquetas admisibles para cumplir con el requerimiento de conformidad de la malla final.

Por ejemplo la aplicación del procedimiento de subdivisión de las regiones 2, 3, 4 y 5 generan tres vértices extras en los límites de la región, ver figura 3.12, de esta manera no es posible subdividir la subregión sin violar el requerimiento de conformidad. Una asignación de etiqueta es inadmisible si contiene uno de los cuatro casos mostrados en la figura 3.13. Para resolver este problema es necesario eliminar los casos ilegales. Este proceso se expone a continuación.

Una subregión sb_i es definida por cuatro vértices $v_{i,j}$, $v_{i,j+1}$, $v_{i,j+2}$ y $v_{i,j+3}$, donde $i = 1, \ldots,$ número de subregiones con $1 \leq j \leq 4$. Los pares de vértices $(v_{i,j}, v_{i,j+2})$ y $(v_{i,j+1}, v_{i,j+4})$ son llamados vértices opuestos de la subregión sb_i.

Sea L un asignador de etiquetas en el conjunto de vértices V, por lo tanto $L: V \to N \cup \{0\}$, además $L(v_{i,j})$ represente la etiqueta

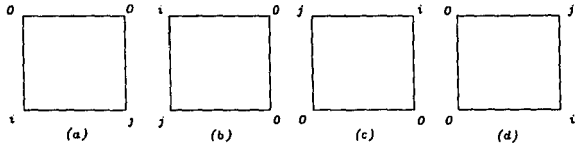


Fig 3.13 Asignamiento de etiquetas ilegales.

de $v_{i,j}$ y la serie de vértices soportados de L sea V_0^L. El complemento de V_0^L en V, $V - V_0^L$, se simboliza por V_1^L. De acuerdo con la definición de extensión admisible, L es un asignador de etiquetas *admisibles* de V si y sólo si *para cualquier subregión* sb_1, *el valor de L es igual a cero para exactamente dos vértices entonces estos son opuestos entre si en* sb_1.

Ahora formalmente, un asignador admisible de etiquetas G en V es llamada una *extensión admisible* de L si:

$$G(v) = \begin{cases} L(v), & v \in V_1^L \\ 0 \text{ ó } 1, & v \in V_0^L \end{cases} \qquad (3.3)$$

Notar que la extensión G de L tal que $G(v) = 1$ para toda $v \in V_0^L$ es también una extensión admisible de L.

A continuación se presenta el algoritmo que se emplea para construir una extensión admisible que minimiza el número de ceros desapareciendo los casos ilegales. Para este fin, se requiere un operador en la asignación de la etiqueta que en lo sucesivo será representado por V. Sean dos asignadores de etiquetas F y G que se definen en V tal que $F(v_{i,j})$ y $G(v_{i,j})$ son cualquiera de los valores 0, $\frac{1}{2}$ ó algún entero positivo. El operador V aplicado entre F y G, es definido como sigue:

$$F \vee G(v_{i,j}) = \max\{F(v_{i,j}), G(v_{i,j})\}, \; v_{i,j} \in V \qquad (3.4)$$

Dos asignaciones especiales de etiqueta L_1 y L_p se presentan a continuación.

$$L_1(v_{i,j}) = \begin{cases} \frac{1}{2} \text{ si el nodo del vértice es impar} \\ 0 \text{ si el nodo del vértice es par} \end{cases} \quad (3.5)$$

$$L_p(v_{i,j}) = \begin{cases} \frac{1}{2} \text{ si el nodo del vértice es par} \\ 0 \text{ si el nodo del vértice es impar} \end{cases} \quad (3.6)$$

Notar que ambas L_1 y L_p son asignaciones admisibles de etiquetas y $L_1 \vee L$ y $L_p \vee L$ son extensiones admisibles de L, ahora se analizará el algoritmo que realiza esta tarea.

1.- Construcción de G_p

1.1 Sea $G_p = L \vee L_p$

Para cada vértice $v_{i,j}$ *de P hacer*

$G_p(v_{i,j}) = L \vee L_p(v_{i,j})$

fin

fin

1.2 Para cada $v_{i,j}$ *tal que* $G_p(v_{i,j}) = \frac{1}{2}$ *hacer*

Si $G_p(v) > \frac{1}{2}$ *para un mínimo de un vértice v adyacente de* $v_{i,j}$

entonces $G_p(v_{i,j}) = 1$

de otra manera $G_p(v_{i,j}) = 0$

fin

fin

fin

2.- Construir G_1

2.1 Sea $G_1 = L \vee L_1$

Para cada vértice $v_{i,j}$ *de P hacero*

$G_1(v_{i,j}) = L \vee L(v_{i,j})$

fin

fin

2.2 Para cada $v_{i,j}$ *tal que* $G_1(v_{i,j}) = \frac{1}{2}$ *hacer*

Si $G_i(v) > \frac{1}{2}$ *para un mínimo de un vértice v adyacente de* $v_{i,j}$
entonces $G_i(v_{i,j}) = 1$
de lo contrario $G_i(v_{i,j}) = 0$
fin
fin
fin
3.- *Construir G*
Si $V_0^{G_i} > V_0^{G_p}$
entonces regresar $G = G_i$
de otra forma regresar $G = G_p$
fin
fin

Algoritmo que construye una extensión admisible (CEA)

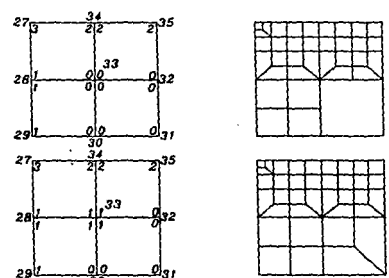


Fig.3.13a Aplicación del proceso de subdivisión sin el algoritmo CEA, superior, y con su aplicación, inferior.

En la figura 3.13a se muestra parte de una red rectangular en la cual se asigno las etiquetas de subdivisión, como se ve estas etiquetas tienen el inconveniente de formar una subdivisión inadmisible.

Es necesario aplicar el algoritmo CEA para eliminar este

inconveniente. En el paso 1, *construcción de* G_p, se lleva a cabo con el empleo del operador V entre L y L_p. Las etiquetas asignadas así como los nodos de la figura 3.13a se muestran en la tabla de valores, columnas dos y uno respectivamente, donde L_p se calcula de acuerdo a la asignación (3.6) y a los nodos de los vértices. Si es impar le corresponde 0 mientras que si es par le corresponde $\frac{1}{2}$, columna tres.

$L(V_{i,j})$	L	L_p	G_p	L_1	G_1	G_p	G_1	G
27	3	0	3	1/2	3	3	3	3
34	2	1/2	2	0	2	2	2	2
35	2	0	2	1/2	2	2	2	2
28	1	1/2	1	0	1	1	1	1
33	0	0	0	1/2	1/2	0	1 *	1
32	0	1/2	1/2	0	0	1 *	0	0
29	1	0	1	1/2	1	1	1	1
30	0	1/2	1/2	0	0	1 *	0	0
31	0	0	0	1/2	1/2	0	0**	0

Tabla de valores de la figura 3.13a aplicando el algoritmo CEA.

Con los valores de L y L_p se aplica el operador V definido por la expresión (3.4), paso 1.1 del algoritmo, de esta manera se toma la etiqueta mayor para el nodo que se analiza, los resultados pueden observarse en la columna cuatro.

A continuación, se analizan las etiquetas de G_p con el valor de $\frac{1}{2}$, paso 1.2, si las etiquetas de los nodos adyacentes al que se analizan son diferentes de cero o de $\frac{1}{2}$ en por lo menos una, se cambiará el valor de $\frac{1}{2}$ por el de 1, como se observa en la columna siete marcado por *. De lo contrario a esta etiqueta nuevamente se le asigna el valor de cero.

Para construir G_1, paso 2 del algoritmo CEA, se aplica el operador V , entre L y L_1, columa dos y cinco respectivamente. L_1 se desarrolla por medio del asignador (3.5), empleando el operador V se lleva a cabo el paso 2.1 del algoritmo los resultados de este proceso se observan en la columna seis de la tabla.

En el paso 2.2, se analizan las etiquetas con valor de $\frac{1}{2}$ y si las etiquetas de los nodos adyacentes es diferente de cero o $\frac{1}{2}$ entonces se le asigna el valor de 1, columna ocho de la tabla marcado por un *, de lo contrario se le asigna el valor de cero. En la figura 3.13a se observa que el nodo 31 tiene en los nodos adyacentes las etiquetas cero, por tal motivo se le asigna el valor de cero, columna ocho renglón marcado con **.

Como ya se menciono el propósito es minimizar el número de cambios o de ceros, así en el paso 3 del algoritmo, se toma como extensión admisible a aquella serie de etiquetas que se aproximen más a la original, L. Al comparar G_p y G_i, columnas siete y ocho, se puede apreciar que en G_i los cambios han sido menores por este motivo se selecciona a G_i como la extesión admisible G, columna nueve, además esta cumple con las condiciones (3.3). Para más información al respecto se puede consultar Cheng (*Cheng et al, 1989*).

Una vez, que se concluyó la aplicación del algoritmo CEA la red cuadrilátera regular esta preparada para el proceso de subdivisión.

3.5.4 subdivisión balanceada.

El algoritmo principal controla dos procesos de subdivisión, balanceada y no balanceada, que se dan de acuerdo a las etiquetas asignadas a los vértices de cada subregión.

El proceso de subdivisión balanceada se efectúa con la rutina *subdivide_2* en la subregión sb = v_1, v_2, v_3, v_4, donde v_n simboliza las etiquetas de los vértices de sb, ver figura 3.14. Para que se realize la subdivisión balanceada, en el algoritmo principal se revisa que un minimo de dos etiquetas de sb sean no cero. Así el proceso genera cuatro subcuadriláteros, sb_n, con sus

respectivas etiquetas, q_n, r_n, s_n, t_n:

$$sb_1 = q_1, q_2, q_3, q_4$$
$$sb_2 = r_1, r_2, r_3, r_4$$
$$sb_3 = s_1, s_2, s_3, s_4$$
$$sb_4 = t_1, t_2, t_3, t_4$$

su distribución se aprecia en la figura 3.14.

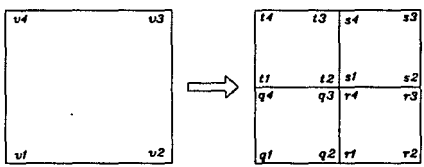


Fig.3.14 Subdivisión balanceada.

Los nuevos valores de las etiquetas son definidos de la siguiente manera:

$$ETIQUETA(q_1) = \max\{0, ETIQUETA(v_1)-1\}$$
$$ETIQUETA(r_2) = \max\{0, ETIQUETA(v_2)-1\}$$
$$ETIQUETA(s_3) = \max\{0, ETIQUETA(v_3)-1\}$$
$$ETIQUETA(t_4) = \max\{0, ETIQUETA(v_4)-1\}$$
$$ETIQUETA(q_2) = ETIQUETA(r_1) = \min\{ETIQUETA(q_1), ETIQUETA(r_2)\}$$
$$ETIQUETA(r_3) = ETIQUETA(s_2) = \min\{ETIQUETA(r_2), ETIQUETA(s_3)\}$$
$$ETIQUETA(s_4) = ETIQUETA(t_3) = \min\{ETIQUETA(s_3), ETIQUETA(t_4)\}$$
$$ETIQUETA(t_1) = ETIQUETA(q_4) = \min\{ETIQUETA(t_4), ETIQUETA(q_1)\}$$
$$ETIQUETA(q_3) = ETIQUETA(r_4) = ETIQUETA(s_1) = ETIQUETA(t_2)$$

las etiquetas centrales de sb se definen como:

$$= \begin{cases} 0 \text{ si } q_2,\ r_3,\ s_4 \text{ y } t_1 \quad \text{son cero} \\ \text{de otra manera} \\ \min\{\text{ETIQUETA}(v) \mid v \in \{q_2,\ r_3,\ s_4,\ t_1\},\ \text{ETIQUETA}(v)>0\} \end{cases}$$

3.5.5 Subdivisión no balanceada.

Como ya se mencionó, el algoritmo principal controla el proceso

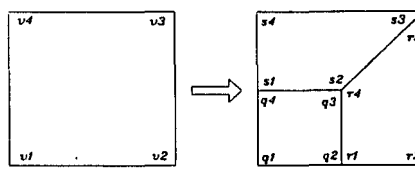


Fig.3.15 Subdivisión no balanceada

de subdivisión no balanceada, revisando las etiquetas de la subregión $sb = v_1, v_2, v_3, v_4$, ver figura 3.15. De esta manera para llevar a cabo el proceso es necesaria la condición de tener exactamente una etiqueta mayor que cero. Con esta condición el proceso *subdivide_1()* se encarga de realizar la subdivisión no balanceada con respecto al vértice con la etiqueta mayor que cero, creando tres cuadriláteros, sb_n, con sus respectivas etiquetas:

$$sb_1 = q_1,\ q_2,\ q_3,\ q_4$$
$$sb_2 = r_1,\ r_2,\ r_3,\ r_4$$
$$sb_3 = s_1,\ s_2,\ s_3,\ s_4$$

cuyos valores se asignan de la siguiente manera:

supongase que v_1 es el vértice con la etiqueta mayor que cero

$$\text{ETIQUETA}(q_1) = \text{ETIQUETA}(v_1) - 1$$
$$\text{ETIQUETA}(q_i) = 0, \qquad i = 2,\ 3,\ 4$$

$$ETIQUETA(r_j) = 0, \qquad j = 1, 2, 3, 4$$
$$ETIQUETA(s_k) = 0, \qquad k = 1, 2, 3, 4$$

3.5.6 El algoritmo principal.

El algoritmo principal tiene el propósito de administrar los diferentes procesos para generar la malla de elementos cuadrangulares. Como inicio un preproceso se encarga de proporcionar los datos necesarios, para iniciar la generación. Este algoritmo se puede dividir en los tres pasos siguientes.

En el primer paso, con los niveles de subdivisión, se asigna a los vértices de cada subregión inicial una etiqueta. Como segundo paso, se comprueba que las etiquetas asignadas no provoquen una subdivisión inadmisible, esto se realiza con la aplicación del algoritmo CEA.

Una vez que se aplicó el algoritmo CEA, se lleva a cabo el proceso de subdivisión, paso tres. Esta subdivisión se controla por el algoritmo principal por el análisis de las etiquetas de cada subregión. De acuerdo a estas etiquetas se realiza la subdivisión balanceada o no balanceada. El proceso termina cuando todas las etiquetas son cero.

3.6 Descripción del algoritmo generador de elementos triangulares.

El algoritmo generador de mallas con elementos triángulares se apoya en el articulo de Deljouie-Rakhshandeh. En este algoritmo por medio de un proceso de suavizado se obtiene una malla con un minimo de triángulos obtusos. Para realizar con rapidez este proceso se seleccionó una estructura de datos que contiene la información de los elementos y nodos adyacentes a cada nodo. También se emplea la descomposición de regiones multiplemente conectadas y la división de regiones no convexas para transmitir la graduación de los elementos de la frontera al interior de la región.

Como ya se mencionó, para obtener resultados confiables el MEF impone ciertas restricciones a las formas de los elementos. En las mallas triangulares, la principal restricción establece que los ángulos internos de los elementos no deben de ser obtusos o muy agudos. Es decir, se pretende que estos se aproximen a los de un triángulo equilátero.

Para el generador de elementos triangulares se tienen los siguientes objetivos: que genere un minimo de triángulos obtusos, rapidez en su operación, simplicidad en su diseño y facilidad en su uso.

La técnica de generar elemento por elemento es atractiva, ya que permite tener cierto grado de control sobre la forma de cada triángulo y además durante y después de la triangulación se suaviza la malla, removiendo muchos de los triángulos obtusos.

El suavizado de la malla es un proceso repetitivo, por esta razón se desarrolló una estructura de datos que contiene la información adyacente tanto de elementos como de nodos de esta

manera fácilmente se localizan los lados que se desplazarán y por lo tanto se incrementa la rapidez de operación del algoritmo.

La técnica de subdividir la región no solo se utiliza para descomponer las regiones no convexas sino también para lograr una graduación adecuada en el tamaño de los elementos triangulares. Para su implementación en el algoritmo se usa la recursividad de los lenguajes modernos como C (capacidad de una rutina para llamarse asi misma), manteniendo la simplicidad en su diseño.

Para iniciar el proceso de discretización, se define la frontera de la región por una serie de lineas rectas, se especifica el sentido y las coordenadas de los nodos iniciales de cada vértice y el tamaño promedio de los elementos en cada extremo de las lineas rectas, con esto la facilidad en su uso se cumple ya que son minimos los datos para el proceso de discretización.

3.6.1 Definiciones.

Se comenzará por definir una región, R, como el área limitada por un plano cuya frontera es ∂R, que se forma por segmentos de lineas rectas semejando a una poligonal. Una región R es multiplemente conectada si contiene por lo menos un hueco.

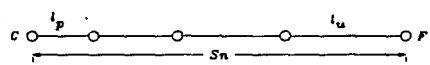


Fig.3.16 Subdivisión de un segmento de linea

En los extremos de las lineas rectas, que definen la frontera de la región, se encuentra un nodo de comienzo y uno de final, C y F, como se muestra en la figura 3.16. También se define en cada extremo el tamaño del elemento seleccionado representado por l_p el

primero y l_u el último. De acuerdo a la longitud total del segmento de línea, S_n, y a los valores de l_p y l_u se calcula el número de nodos, n, y su distribución, r, que se insertarán entre los nodos C y F. Esta tarea se realiza por medio de una progresión geométrica:

$$S_n = \sum_{i=1,\ n} ar^{i-1} \qquad (3.7)$$

donde a, r, y n son seleccionados de tal manera que:

$$a = l_p \quad \text{y} \quad ar^{n-1} = l_u \qquad (3.8)$$

se transforma a (3.7) en una forma más adecuada:

$$S_n = \frac{a_1 - ra_n}{1 - r} \qquad (3.9)$$

se despeja a la distribución, r, en (3.9) y al número de nodos por insertar, n, en (3.8) se tiene:

$$r = \frac{a_1 - S_n}{a_n - S_n} \qquad (3.10)$$

$$n = \frac{\log\left[\frac{a_n}{a_1}\right]}{\log(r) + 1} \qquad (3.11)$$

para más información acerca de las expresiones (3.10) y (3.11) consultar a Lehmann.

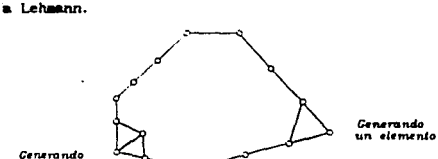


Fig.3.17 Generación de elementos.

De esta forma la frontera queda definida por las lineas rectas. Si la región es multiplemente conectada, tenga uno o varios huecos en su interior, es subdividida en varias simples y las regiones simples son divididas en varias convexas para posteriomente ser trianguladas.

En una región convexa los ángulos interiores son examinados, si el ángulo es menor a uno preseleccionado un triángulo se crea, de otra manera dos triángulos se crean (ver figura 3.17). Cada triángulo que se crea introduce nuevos ángulos que serán examinados, el proceso de triangulación continuará hasta cubrir toda la zona. Los triángulos deben de ser tan cercanos, como sea posible, al equilátero.

El algoritmo emplea tres tipos de listas adyacentes: nodo-nodo adyacente, nodo-elemento adyacente y elemento-nodo adyacente.

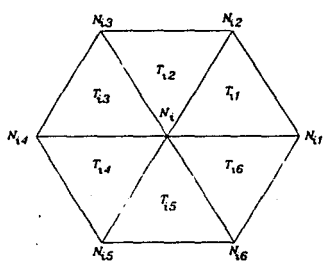


Fig.3.18 Orden de listas nodo-nodo adyacente y nodo-elemento adyacente.

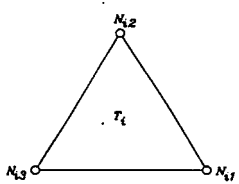
La lista nodo-nodo adyacente es una ordenación circular de nodos en el sentido contrario a las manecillas del reloj, como se muestra en la figura 3.18, la definición de ésta en un arreglo es:

$$Adj[N_i, j] = N_{i,j}, \qquad j = 1, \ldots, r(N_i)$$

donde $r(N_i)$ es el número total de nodos adyacentes al nodo N_i, también se define un apuntador (que es una variable que almacena la dirección del arreglo), $1 \le P(N_i) \le r(N_i)$, tal que si $Adj[N_i, P(N_i)] = N_{i,4}$ entonces $Adj[N_i, P(N_i)+1] = N_{i,5}$. La lista nodo-elemento adyacente es construida en la misma forma que la lista nodo-nodo adyacente (ver figura 3.18):

$$Tri[N_i, j] = T_{i,j}, \qquad j = 1, \ldots, r'(N_i) \quad \text{con } 1 \le P'(N_i) \le r'(N_i)$$

donde $r'(N_i)$ es el número total de elementos adyacentes y $P'(N_i)$ el apuntador del arreglo.



Fug.319a Nodo-elemento adyacente

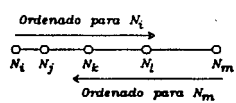


Fig 3.19b Ordenado nodal

La lista elemento-nodo adyacente es simplemente:

$$Elem[T_i, j] = N_{i,j}, \qquad j = 1, \ldots, 3$$

la estructura de ésta se muestra en la figura 3.19a.

Todas las listas adyacentes se construyen durante el proceso de generación de la malla, evitando tener que preseleccionar las dimensiones de los arreglos para cada región.

Con la ayuda de las listas adyacentes la región puede ser referida con un nodo fronterizo, N. Pero además es necesario

definir la función de transferencia $\Gamma()$, tal que:

$\Gamma^0(N) = N$,

$\Gamma(N) = Adj[N, P(N)]$, $\Gamma^{-1}(N) = Adj[N, P(N)-1]$,

$\Gamma^2(N) = Adj[\Gamma(N), P(\Gamma(N))]$, $\Gamma^{-2}(N) = Adj[\Gamma^{-1}(N), P(\Gamma^{-1}(N))-1]$,

y en general

$\Gamma^n(N) = Adj[\Gamma^{n-1}(N), P(\Gamma^{n-1}(N))]$,

$\Gamma^{-n}(N) = Adj[\Gamma^{1-n}(N), P(\Gamma^{1-n}(N))-1)]$.

con la función $\Gamma()$, un apuntador y la lista nodo-nodo adyacente se puede recorrer la frontera de la región en cualquier dirección, sin la necesidad de algún arreglo especial que se tenga que cambiar para cada subregión.

También es necesario definir un borde recto, **E(P,Q)** con la orientación del punto **P** al punto **Q**. De esta manera los nodos fronterizos de **R** cumplen con la siguiente expresión:

$$\partial R = \bigcup_{r=1,n} E(\Gamma^{r-1}(N), \Gamma^r(N)) \qquad (3.12)$$

Donde **N** es un nodo fronterizo de la región, y **n** es el número de nodos de la frontera tal que satisfacen la siguiente relación:

$$\Gamma(N)^n = N \qquad (3.13)$$

es decir si se emplea la función de transferencia con el número de nodos como lugares que se desean recorrer y un nodo de la región se llegará a este mismo nodo.

De esta manera la frontera, ∂R, es un circuito que puede ser recorrido en el sentido de las manecillas del reloj

procediendo de $\Gamma^{r}(N)$ a $\Gamma^{r+1}(N)$. Los nodos que satisfacen la relación (3.12) son llamados nodos activos.

Los nodos distribuidos en la frontera de la región inicial, o en la frontera compartida por dos subregiones, se activan por un simple procedimiento de ordenado nodal, ver figura 3.19b. Con este proceso se puede cambiar el sentido de un borde, cambiar de región activa o apartar una subregión del resto de la región.

El procedimiento altera la lista nodo-nodo adyacente de los nodos distribuidos en una linea y crea una dirección implicita. Para llevar a cabo el proceso de ordenado nodal, un mínimo de tres nodos son necesarios. Para ordenar los nodos en la figura 3.19b de N_i a N_e, los apuntadores de N_i y N_j de la lista nodo-nodo adyacente son una serie tal que:

$$\Gamma(N_i) = N_j, \text{ y } \Gamma^{-1}(N_j) = N_i \qquad (3.14)$$

así que, $N_k = \Gamma(N_j)$ y un proceso similar se realiza para el nodo N_e.

El ordenado de los nodos fronterizos para una región o una subregión crea una serie de nodos activos:

$$SRP = N, \Gamma(N),\ldots,\Gamma^{n-1}(N) \qquad (3.15)$$

donde N es un nodo fronterizo y satisface la relación (3.12). La región que limita este conjunto de nodos es llamada una región activa.

Para obtener una región activa, no es necesario mantener los nodos que la limitan en un arreglo especial, basta con un nodo activo y la relación (3.12).

3.6.2 Subdivisión de regiones no convexas

Las regiones no convexas realmente no constituyen dificultad alguna para el proceso de triangulación, sin embargo para lograr una malla con elementos distribuidos gradualmente se recurre a la subdivisión de éstas.

La región que se discretiza se define como **R(N)**, donde **N** es un nodo activo, **R(N)** es dividida por medio de líneas rectas, llamadas conectores, donde cada conector une a dos nodos de **R(N)**. Para construir un conector todos los ángulos interiores de **R(N)** mayores de 180• se identifican, el vértice con el ángulo interior mayor es el primer nodo del conector, nodo C_1 en la figura 3.20a. En seguida se calculan los nodos visibles para C_1 de entre los cuales se escoge a Q_1 por medio de la espresión:

$$\alpha_l = \min(\alpha_1, \alpha_2, \alpha_3, \alpha_4) \qquad (3.16)$$

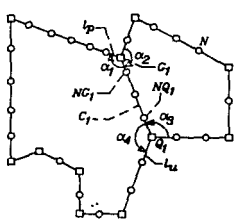


Fig.3.20a Subdivisión de una región por medio de un conector.

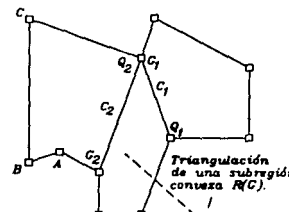


Fig 3.20b División de una región no convexa y la triangulación de una de sus subregiones.

donde l = 1, ..., número de nodos visibles. Por medio de (3.16) se calculan todas las α_l mínimas y se toma como nodo Q_1 al que tenga el valor máximo de las α_l .De esta manera se forma el conector C_1, como se muestra en la figura 3.20a.

En el conector se insertan nuevos nodos tal que su número y distribución se calcula por medio de la progresión geométrica

(3.7) y el tamaño de los elementos adyacentes en G_i, l_p, y en Q_i, l_u. También el conector, C_i, es ordenado de G_i a Q_i, y se le identifica como un arreglo con cuatro componentes:

$$C_i = [G_i,\ NG_i,\ Q_i,\ NQ_i] \qquad (3.17)$$

donde i es el índice del conector y NG_i y NQ_i son los nodos adyacentes a G_i y Q_i, respectivamente como se muestra en la figura 3.20a.

En la figura 3.20a el ordenado nodal se realiza de G_i a Q_i, la región $R(G_i)$ será la zona de la izquierda, y si el ordenado es realizado de Q_i a G_i, la región $R(G_i)$ será la zona de la derecha. De esta manera una región activa puede ser definida por un ordenado nodal. Esta facilidad reduce el número de argumentos de control requeridos para manejar varias subregiones.

El algoritmo tiene dos argumentos N un nodo activo y n que es el índice del conector. Al iniciar el procedimiento, n se inicializa a cero ya que ningún conector ha sido generado. Después de localizar el ángulo interior mayor, el índice del conector, n, se incrementa y un conector, C_n (C_i figuras 3.20a y 3.20b), se construye al satisfacer la condición de la expresión (3.16). Los nodos que se generan por la progresión (3.7) son ordenados de G_n (G_i) a Q_n (Q_i) de esta manera la región inicial es automáticamente descompuesta, y la subregión activa es ahora $R(G_n)$ ($R(G_i)$).

La nueva subregión activa es recursivamente examinada para comprobar que sea convexa. Si es así, entonces $R(G_n)$ ($R(G_i)$) se discretiza ver figura 3.20b, zona sombreada. Después de completar la triangulación, la región discretizada es removida del dominio inicial simplemente reordenando los nodos adyacentes en C_n de Q_n a G_n además de esto se activan los nodos fronterizos de la nueva subregión.

Con referencia a la figura 3.20b, después de reordenar C_2 de Q_2 a G_2 $R(G_1)$ la nueva subregión referida será la poligonal con vértices en Q_2, G_2, A, B, y C. Ahora el índice del conector, n, decrece y la nueva subregión tiene un proceso similar al de $R(G_1)$. Así el procedimiento termina cuando n recupera el valor inicial de cero.

3.6.3 Descomposición de regiones multiplemente conectadas

El algoritmo de generación de mallas con elementos triangulares solo puede tratar regiones simples. Las regiones multiplemente conectadas deben de ser descompuestas en subregiones simples.

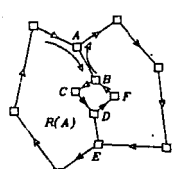


Fig.3.21a Activado de una subregión multiplemente conectada.

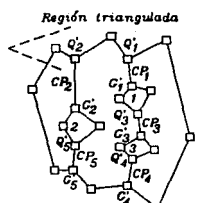


Fig.3.21b Subdivisión y triangulación de una región multiplemente conectada.

El proceso de recursividad que se empleó en la subdivisión de regiones no convexas es la misma que se utiliza en las regiones multiplemente conectadas. Nos referiremos a ∂R como la frontera externa de la región, y a ∂h como a la frontera de los huecos, donde sus nodos se numerán secuencialmente y se activan de tal manera que en ∂h se recorreran en la dirección contraria a las manecillas del reloj, mientras que en la frontera externa ∂R los

nodos activos se mueven en el sentido de las manecillas del reloj, ver figura 3.21a). Se localizan los vértices más alto y más bajo de cada hueco entonces se conectan a los vértices de ∂R por medio de líneas rectas que se llaman *conectores parciales*. En la figura 3.21a **AB** y **DE** son conectores parciales que unen a los vértices, **A** y **E** de ∂R, através de una *cadena* la cual contiene las dos líneas de ∂h.

Los conectores parciales son respectivamente ordenados de **A** a **B** y de **D** a **E**. La cadena correspondiente, por lo tanto pasa por los puntos **A**, **B**, **C**, **D** y **E** en orden consecutivo y la región **R(A)**, es la de la izquierda. Si los conectores parciales son reordenados en dirección reversiva la cadena pasará por **E**, **D**, **F**, **B** y **A**, y **R(A)** será la región de la derecha del hueco.

Para regiones con huecos múltiples, el preproceso involucra conectar el vértice mayor del hueco, ∂h_i, a cualquier nodo de ∂R o a un vértice de otro hueco. Esto se realiza observando primero todos los vértices visibles para G_i, entonces se construye el mejor conector parcial entre G_i y tal vértice visible. Así los conectores llevan la información nodal del hueco y del punto final al que se unen, como se muestra en el siguiente arreglo:

$$CP_i = [G'_i,\ MG'_i,\ Q'_i,\ MQ'_i,\ j] \qquad (3.18)$$

donde **i** indica que el origen del CP_i esta en el hueco **i**, G'_i pertenece a ∂h_i, **j** indica que termina en el hueco **j**, Q'_i pertenece a ∂h_j, y a **j** se le da el valor de cero cuando el conector parcial termina en ∂R.

Al terminar el preproceso, el número de conectores parciales es igual al número de huecos, ver figura 3.21b donde los conectores parciales CP_1 CP_2 y CP_3 fueron creados, transformando la región multiplemente conectada a una simplemente conectada. El algoritmo de descomposición transfiere el control al algoritmo de

división de regiones no convexas, quien lleva a cabo la subdivisión de la región recursivamente y construye una lista de huecos:

$$H = [\lambda_1, \lambda_2, \ldots, \lambda_k] \tag{3.19}$$

donde k es el número de huecos, λ_1 contiene un número entero que corresponde al ordenamiento de los huecos tal que:

$$y(b_{\lambda_1}) < y(b_{\lambda_2}) < \ldots < y(b_{\lambda_k}) \tag{3.20}$$

donde b_i es el vértice más bajo del hueco i, y $y(b_i)$ es el vértice más alto . Tomando en cuenta (3.20), la lista de huecos para la figura 3.21b es:

$$H = [3, 2, 1]$$

como se aprecia el orden es de menor a mayor.

Después del preproceso, la discretización de la región se inicia con la revisión de los huecos de la región y se construyen los demás conectores parciales, así se toma el vértice menor b_{λ_1} y un conector parcial se construye entre este vértice y uno de ∂R. Cuando este proceso se lleva a cabo en la región de la figura 3.21b se completa la primera cadena, Cd_1, que la divide en dos subregiones. La cadena creada contiene el índice de todos los conectores que la forman por lo que Cd_1 de la figura 3.21b es:

$$Cd_1 = [4, 3, 1] \tag{3.21}$$

los conectores parciales son ordenados de Q_i' a C_i', con esta operación se crearán nuevos circuitos a lo largo de la cadena. La nueva subregión, zona a la izquierda de Cd_1, se examina por el mismo proceso, si ésta contiene un hueco una nueva cadena es construida y el proceso se repite hasta que la subregión

resultante no contenga más huecos.

Una vez que la subregión no contiene más huecos es procesada por los algoritmos de división y triangulación. Esto se muestra en la figura 3.21b donde la zona sombreada es una subregión simple no convexa. Después de la triangulación de una subregión, todos los nodos de los conectores parciales que forman a la última cadena se reordenan. El reordenamiento reversivo lleva consigo en la orientación de la cadena el definir una nueva subregión y un nuevo circuito.

En la figura 3.21b, después de reordenar PC_5 y PC_2 la nueva subregión activa será la zona que se halla entre Cd_1 y Cd_2, también la subregión sombreada es removida y como Cd_2 no es más necesaria el índice global de las cadenas es disminuido en uno.

El algoritmo de descomposición acepta dos argumentos: N, un nodo activo, y r el índice de las cadenas que se incrementa conforme se crean las cadenas y disminuye cuando estas son reordenadas, por lo cual cuando r obtiene el valor inicial, de cero, se da por terminado el proceso de descomposición.

3.8.4 Triangulación de regiones convexas.

En lo sucesivo una región activa será referida como R(N) y su frontera como $\partial R(N)$, donde N es un nodo activo. En el proceso de triangulación algunos nodos activos son hechos inactivos y nuevos nodos activos son generados, este proceso mantiene a la región sin triangular activa. Así, la región R(N), mostrada en la figura 3.22 es activa en evolución la cual disminuye conforme los triángulos son generados. Además, un triángulo puede ser generado en cualquier lugar de $\partial R(N)$, e inmediatamente después la región restante se convierte en activa, de esta manera uno o dos elementos son separados de la región.

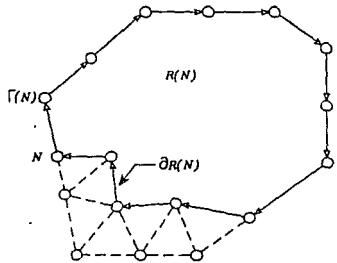


Fig.3.22 Evolución de una región activa, R(N), durante la triangulación.

La estrategia para triangular es comenzar por un nodo activo N_i y calcular el ángulo interior correspondiente, θ_i, de acuerdo a la magnitud de θ_i dos tipos de operación tienen lugar: la primera consiste en la inserción de un punto por medio del algoritmo *bisectar*; en el segundo, se genera un triángulo con la unión de dos puntos nodales que cierran un ángulo este proceso se realiza por medio del algoritmo *cerrar*, figura 3.23. En ambos casos las listas adyacentes de dos o más nodos son alteradas, esto se realiza mediante el algoritmo *alterar*.

El algoritmo *alterar* se encarga de cambiar la lista nodo-nodo adyacente para ambos nodos N_i y N_j tal que:

$$\Gamma(N_j) = N_i \qquad \text{y} \qquad \Gamma(N_i) = N_j$$

Con referencia a la figura 3.23 para ambos casos cerrar y bisectar las alteraciones de la lista *nodo-nodo* adyacente son llevadas a cabo en la secuencia siguiente:

para cerrar: **ALTERAR**(N_k, N_j) (3.22)

y para bisectar

$$\left.\begin{array}{l} \text{ALTERAR}(N_i, N_a) \\ \text{ALTERAR}(N_a, N_j) \\ \text{ALTERAR}(N_k, N_a) \end{array}\right\} \qquad (3.23)$$

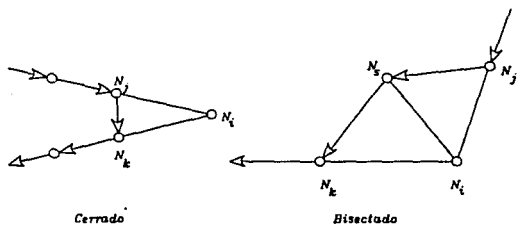


Fig.3.23 El cambio de dirección esta marcado por el proceso alterar que modifica la lista nodo-nodo adyacente durante la triangulación.

Por lo tanto, el algoritmo *bisectar* crea un nuevo nodo activo, N_a, y transforma a N_i en un nodo inactivo. Similarmente el algoritmo de cerrar hace a N_i inactivo (tal que $\Gamma(N_j) \neq N_i$) mientras que N_j y N_k permanecen activos.

Por cada nuevo triángulo que se genera la lista elemento-nodo adyacente se incrementa, también se tiene una secuencia similar para adicionar y alterar la lista nodo-elemento adyacente. Para completar el proceso de bisectar o el de cerrar, el generador de mallas se mueve hacia el siguiente nodo activo N_k y se continúa con el proceso. El último triángulo de la malla es el único con tres nodos activos, el proceso de triangulación se concluye cuando este triángulo se genera.

En un nodo activo N_i, el ángulo interior θ_i se calcula, y dos ángulos límite β_1 y β_2 se preseleccionan, tal que $\beta_1 < \beta_2$, si $\theta_i < \beta_i$ entonces el algoritmo *cerrar* se emplea y un triángulo se genera

con la unión de los dos nodos adyacentes al nodo activo. Si, $\beta_1 < \theta_i < \beta_2$, entonces se llama al algoritmo *bisectar*, con lo cual un nuevo nodo es insertado y dos triángulos se generan.

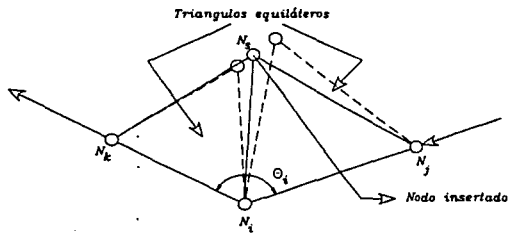


Fig.3.24 Inserción de un nodo.

El algoritmo *cerrar* es un proceso puramente de cambio en las listas adyacentes, pero para *bisectar*, donde se insertó un nodo, se necesitan algunos conceptos de distancia y ángulo, como se muestra en la figura 3.24, dos triángulos equiláteros son construidos entre el ángulo θ_i que encierra a dos bordes. El punto medio de los dos triángulos es tomado como el nuevo nodo, N_s, éste es aceptado o rechazado de acuerdo al siguiente criterio: se calcula a V_N, una subserie de los nodos activos ,∂RP , como el conjunto de nodos visibles para N_i, con éstos, N_s se acepta si la distancia entre éste y cualquiera de los nodos $N_N \in V_N$ satisface la siguiente relación:

$$\text{Dist}(N_s, N_N) > \alpha \, \text{Avrg}(N_N); \qquad N \in V_N \tag{3.24}$$

Donde Dist es la función distancia, y α es una constante real que vale 0.5 y

$$\text{Avrg}(N_N) = \{\text{Dist}(N_N, \Gamma(N_N,1)) + \text{Dist}(N_N, \Gamma(N_N, \Gamma(N_N,-1))\} \tag{3.25}$$

De otra manera N_s es rechazada y el generador se mueve al siguiente nodo activo $N_k = \Gamma(N_i)$, ver figura 3.25. Si ninguno de los nodos activos de ∂RP son adecuados para los algoritmos *cerrar*

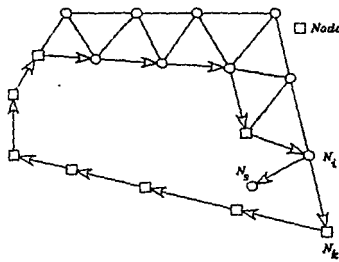


Fig.3.25 Se examina al nodo N_s para ser aceptado.

y ***bisectar*** entonces se divide a la región por medio del algoritmo de división, de esta manera se crearán nuevos ángulos. Si la primera subregión tampoco es adecuada para llevar a cabo los procesos de ***bisectar*** o *cerrar*; la subregión es nuevamente dividida por el algoritmo de *división*.

3.6.5 Suavizado de la malla.

Comunmente, los algoritmos de generación de mallas con elementos triángulares, al terminar el proceso de discretización para lograr una mejor aproximación de los elementos a la forma equilátera, le dan un acabado o suavizado a ésta como último paso. En el caso de este algoritmo ese paso se extiende al aplicarlo durante el proceso de triangulación.

Se dan los acabados a la malla de dos maneras, en caso de nodos inactivos moviendo el nodo al centro gravitacional del

polígono que se creó por sus nodos adyacentes. Para nodos activos, se construyen triángulos equiláteros usando sus nodos adyacentes, ver figura 3.26, y entonces se mueve el nodo activo al centro

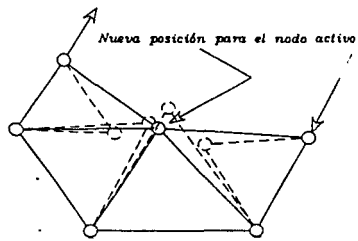


Fig.3.26 Suavizado de malla para un nodo activo.

gravitacional de los vértices de estos triángulos equiláteros. El proceso de suavizado solo se utiliza dos veces durante el mallado, lo cual es suficiente para nuestros propósitos.

Sin las listas adyacentes, el proceso de acabado puede ser computacionalmente costoso. La rapidez esta dada por la facilidad con la cual se puedan identificar los nodos adyacentes de cualquier nodo. Esta información esta disponible simplemente tomando la lista nodo-nodo adyacente.

DESCRIPCION DEL SISTEMA DE GENERACION AUTOMATICA DE MALLAS.

De los algoritmos analizados en el capitulo anterior se han seleccionado dos, uno que genera mallas con elementos cuadrangulares y el otro con elementos triangulares. Estos algoritmos han sido implementados en un programa gráfico que automáticamente genera estas mallas. Primeramente, se analizan las caracteristicas del lenguaje C que se utilizó para programar estos algoritmos, y en segundo lugar se detalla la arquitectura de los principales módulos que forman al programa de generación automática.

4.1 El lenguaje C.

El lenguaje de programación C fue diseñado e implementado por Dennis Ritchie en 1972 en los laboratorios Bell. C fue desarrollado para programadores de sistemas quienes necesitan un lenguaje de alto nivel, como Fortran o Algol, con la eficiencia de un ensamblador de bajo nivel.

Decir alto o bajo nivel no necesariamente implica ventaja o desventaja, sólo se refiere a ciertas caracteristicas de cada lenguaje. Asi un lenguaje de ensamblador esta encaminado a pensar en términos del hardware de las máquinas mientras que para un lenguaje de alto nivel permite desarrollar el código fuente en

terminos más aproximados al problema que el programador quiere resolver.

C es, relativamente, un lenguaje de bajo nivel pero tiene las caracteristicas de un lenguaje de alto nivel y esto se puede notar en el número de operadores y la facilidad con que se pueden administrar todos los recursos de una PC.

Si se mira por primera vez un código fuente y se esta familiarizado únicamente con un lenguaje de alto nivel, puede pensarse que es complicado por su aspecto extraño. Sin embargo, no es asi, algunas caracteristicas que se mencionarán en los párrafos siguientes, determinarán el porque C es un lenguaje adecuado para satisfacer las necesidades especiales que el MEF requiere.

4.1.1 Características.

Con el uso creciente de las computadoras se ha tenido la necesidad de desarrollar software para satisfacer diferentes tareas en las que estas máquinas pueden ayudar al usuario. En los últimos años, los usuarios se han inclinado por el lenguaje C para desarrollar el software que necesitan, ya que brinda caracteristicas que otros lenguajes no ofrecen, tales como:

Bloques estructurados.- Los programas en C constan de bloques de sentencias llamados funciones. Cada función esta claramente definida y realiza una tarea especifica. Asimismo con esta estructuración el programa se puede escribir y leer rápidamente.

Rapidez.- Los programas en C son ejecutados con una mayor rapidez en relación con otros lenguajes. Esto es muy marcado cuando se emplea en los sistemas operativos MS-DOS y UNIX, ya que estos han sido desarrollados en lenguaje C.

Versatilidad.- C es usado para escribir, virtualmente, todo tipo de programas tales como sistemas operativos, compiladores, ensambladores, editores, aplicaciones a programas, animaciones gráficas, programas de negocios y aplicaciones científicas.

Portabilidad.- Un programa escrito en C para una computadora puede ser transportado a otra con pocos cambios o inclusive sin ninguno.

Código compacto.- El tamaño del programa generado en C es, generalmente, menor que el obtenido en otro lenguaje.

Por las características citadas el lenguaje C ha empezado a emplearse con mayor frecuencia en diferentes áreas para desarrollar programas aplicados a la ingeniería.

4.1.2 Funcionamiento.

Un breve repaso de C afirmará lo dicho en párrafos anteriores obtiendo un conocimiento elemental que permita comprender el código fuente del anexo A, pero sin llegar a ser un curso de lenguaje C, para más información consultar a Kassab o Schildt. Iniciaremos con el análisis de un código fuente en este lenguaje.

Como se muestra en la figura 4.1, un programa en C consiste de un preprocesador directivo, de declaraciones de variables y funciones, del cuerpo de una función principal (*main()*), y de comentarios y otras funciones.

Una ventaja única de C es el *preprocesador directivo* que es un proceso iniciado antes que el archivo fuente comience a ser compilado, es decir antes de ser transformado de un código fuente a un lenguaje máquina.

El preprocesador realiza varias tareas importantes tal como incorporar el contenido de otro archivo en el programa C (con el directivo *#include*). Es común que un compilador contenga una librería de funciones estandar y la manera de tener acceso a ellas es por medio de este directivo. También en este preproceso se puede condicionar la adición de archivos empleando los directivos *#if* y *#else*.

```
#include<stdio.h>                                  Preprocesador
#include<math.h>                                   directivo
#define     PI 3.141592654
#define     CUADRADO(x) ((x)*(x))

double area;                                       declaración de
double area_de_circulo(double);                    variables y
                                                   funciones

main()                                             función principal
{
     double radio;
     printf("\n radio del circulo?\n")
     scanf("%lf", &radio);                         cuerpo de
     if(radio <= 0) exit(0);                       una función
     area = area_de_circulo(radio);                o
     printf("Area de circulo, con radio            sentencias
             %f = %f\n", radio, area);
}
/* ---------------------------------*/             comentarios

double area_de_circulo(doble r)
{                                                   otras
     return(4.0 * PI * CUADRADO(r));                funciones
}
```

fig.4.1 Estructura general de un programa en *C*.

El directivo *#define* se utiliza para reemplazar una serie de caracteres por otros que corresponden a una misma variable o un arreglo. Además, se emplea en pequeñas funciones, como se muestra en la figura 4.1, donde los paréntesis extras alrededor de los parámetros son necesarios para una sustitución adecuada. A esa clase de directivos también se les llama definiciones macros, y se utilizan frecuentemente para mejorar la velocidad de los programas y se reduce el tamaño del código.

La *declaración* de variables y funciones es de acuerdo al tipo de datos que se almacenen o manejen durante el programa. Todas las variables deben de ser declaradas antes de usarse, especificando la clase de almacenamiento *(extern, auto, static* y *register)* y el tipo de la variable *(entero: int; punto flotante: float* y *caracteres: char).* Las declaraciones de variables y funciones hechas fuera del cuerpo de la función son consideradas *globales,* esto significa, que se puede tener acceso a ellas desde cualquier función y que su tiempo de " vida " será mientras dure el programa.

Las variables que se declaran dentro de una fución tienen " vida " sólo mientras que ésta es llamada. Así, el valor almacenado se perdera cuando termine el objetivo de la función y el espacio de memoria ocupado estará disponible para otra variable.

Analizando la forma de una declaración como la mostrada en la figura 4.1, se puede observar que tiene la siguiente anatomia:

clase-de-almacenamiento tipo-de-dato nombre-de-la-variable;

Dentro de las declaraciones el *nombre-de-la-variable* no solo se refiere a un número de caracteres que definen a la variable sino que también se incluyen: el *arreglo* (agrupación de elementos de un mismo tipo), el *apuntador* (almacena la dirección de una variable), la *estructura* (es una colección de datos de tipo mixto), la *función* (el tipo de dato que regresa al ser llamada) y la *union* (son datos de diferente tipo pero que comparten la misma área de memoria).

El *cuerpo de una función* contiene declaraciones, expresiones y sentencias que son locales. Las sentencias son aquellas que controlan el flujo de la ejecución y el uso de las variables previstas por las expresiones. Una expresión esta formada por

operadores, variables, y llamadas a funciones.

C tiene una serie de operadores para la comparación aritmética y lógica y también para asignar valores, como otros lenguajes, pero además provee dos operadores poco usuales que son para el incremento y decremento de variables. El operador de incremento ++ adiciona 1 a la cantidad sometida a éste y el operador -- substrae 1 de la misma manera. Estos operadores pueden ser usados como prefijos o posfijos a una variable, de esta forma para el incremento en una unidad es *++variable* o *variable++*. Esto se utiliza comunmente en los ciclos *for*, por ejemplo: *for(i=1; i<=10; ++i)* y *for(i=1; i<=10; i++)*. En el primer caso, *i* se incrementa antes de emplearse, de esta manera tiene un valor inicial de dos mientras que en el segundo caso, es incrementada después de ser usada iniciando el ciclo con un valor de uno. C incluye una forma de asignación para actualizar una variable basada en un valor común por ejemplo la expresión *v+=1* es equivalente en otros lenguajes a *v = v + 1*.

C tiene operadores condicionales (?,:), que evalúan y asignan a verdadero o falso. C permite múltiples asignamientos en una sentencia con el operador coma (,) el cual combina dos expresiones en una, por ejemplo: *for(i=0, j=3; i<8; i++, j++)*. Estos operadores, no solo hacen sentencias compactas sino que también incrementan la rapidez de ejecución.

C provee una serie de sentencias condicionales, ciclos y transferencias de dirección. La sentencia estandar de toma de decisión *if-else* es similar a la empleada en Fortran pero C ofrece otros tipos de ciclo.

El ciclo condicionado puede ser declarado usando al principio de éste: *while*, o al final usando *do-while*. C provee un ciclo *for* en el cual las variables se inicializan al principio del ciclo, de la misma manera que en el ejemplo anterior.

C provee también tres sentencias de transferencia de control incondicional: *break, continue*, y *goto*. Que se utilizan cuando es necesario salir de un ciclo sin comprobar sus condiciones. Esta construcción permite al programador controlar el flujo del programa. Aunque la sentencia *goto* existe es poco común su utilización ya que se ha establecido que no hay situaciones de programación que requieran su uso, además el código fuente tiende a volverse confuso e ilegible.

El fin de una sentencia está marcado por un punto y coma. Adicionalmente con el propósito de facilitar la lectura del programa fuente, un desplazamiento adecuado hacia la derecha, sangría, se realiza en cada línea según su relación entre ellas.

Como en otros lenguajes, se pueden realizar *comentarios* durante la edición del código fuente que describan el objetivo de una sentencia o bloque de sentencias, donde el comentario se establece entre /*....*/ como se indica en la figura 4.1.

En general los programas en C están formados por *otras funciones* aparte de la principal (*main()*) que son equivalentes a las subrutinas en Fortran. La ejecución del programa comienza en la función principal y transfiere el control hacia las otras funciones. El inicio y final de las funciones esta representado por las llaves "{" y "}", respectivamente. Una característica de las fuciones es la *recursividad* que consiste en la posibilidad de llamarse así mismas y que se empleará en los algoritmos que se mostrarán.

Una función en C se accesa por su nombre, seguido por un paréntesis, dentro del que se listan sus argumentos sin la necesidad de la declaración *CALL* utilizada en Fortran.

En general, estas son algunas características del lenguaje de

programación C que ayudarán a entender el pseudocódigo en que estan desarrollados los algoritmos descritos en el capitulo 3.

4.2 Organización del sistema gráfico.

El *sistema gráfico de generación de mallas para el método del elemento finito*, esta formado por los dos algoritmos descritos en el capitulo tres, uno generá elementos cuadrangulares y el otro elementos triangulares. El usuario puede seleccionar el tipo de elementos para el problema que desea resolver. A continuación se describen los algoritmos empleados en el sistema gráfico con un pseudocódigo en lenguaje C que da a conocer la manera en que se implementaron las técnicas y procesos del capitulo 3. Para más detalle al respecto en el anexo A se presenta el código fuente de estos algoritmos.

Veamos primero los módulos que forman al programa en C de los elementos cuadrangulares:

Leer().- El objetivo de este proceso es proporcionar los datos necesarios para generar la malla con elementos cuadrangulares, para ello la región debe dividirse en subregiones, formando una red cuadrilátera regular como se explicó en el inciso 3.5.1; se definen los nodos con sus respectivas coordenadas que componen a cada subregión. Estos datos son almacenados en los arreglos adecuados para su utilización posterior.

```
leer()
{   introducir las coordenadas de cada nodo;
    introducir los nodos iniciales de cada elemento;
    almacenar los datos para su utilizacion posterior;
}
```

Etiqueta_asignada().- Una vez que los datos iniciales son introducidos y almacenados, se definen los niveles de subdivisión de cada subregión y se asigna una etiqueta de subdivisión a los

nodos iniciales, como se describe en el inciso 3.5.2. El algoritmo, elige el nivel de subdivisión máximo de las subregiones que comparten un nodo, si es sólo una subregión, se toma este nivel como máximo. El nivel de subdivisión máximo es la etiqueta asignada a los vértices de las subregiones que comparten al nodo.

Para el proceso anterior es necesario identificar cuales son las subregiones que comparten un mismo nodo y almacenarlas con el nivel máximo de subdivisión para este nodo. El proceso se lleva a cabo por medio del procedimiento *etiqueta_asignada()*.

```
etiqueta_asignada()
{
    introducir el nivel de subdivisión de cada subregión;
    definir memoria dinámica;
    for(i=1; i<=número de nodos; i++)
    {
        for(j=1; j<=número de elementos; j++)
            for(k=1; k<4; k++)
            {
                buscar los elementos que comparten
                                    un mismo nodo;
                buscar el nivel de subdivisión máximo;
            }
        for(k=0; k<número de elementos que comparten
                  al nodo i;                          k++)
        {
            asignar etiquetas a los vértices del nodo i;
                                          /* ver fig.3.11 */
        }
    }
    for(i=1; i<=número de elementos; i++)
    {
        inicializar arreglo para nodos adyacentes;
    }
    for(i=1; i<= número de nodos; i++
    {
        buscar los nodos adyacentes a cada nodo i;
    }
} /* fin de etiqueta_asignada */
```

Extension_admisible().- Durante el proceso de discretización se realizan una serie de subdivisiones que son controladas únicamente por las etiquetas asignadas a los vértices, por esta razón se debe

tener especial cuidado con estos valores. En el inciso 3.5.3. se define una extensión admisible, donde se revisa que las etiquetas asignadas a los vértices sean adecuadas, ver figura 3.14, ya que como algunos bordes son compartidos por dos subregiones es necesario que los nodos generados en estos bordes sean compatibles con las dos subregiones. Además, es necesario que los nodos adyacentes a un nodo par sean impares y viceversa. Esta última condición se debe tomar en cuenta al introducir los datos en la función *leer()*.

```
extension_admisible()
{
    for(i=1; i<= número de nodos; i++)
    {
        if(i es par)
        {
            if(si la etiqueta par asignada es cero)
            {
                tomar para el grupo par el valor de 0.5;
                tomar para el grupo impar el valor de 0.0;
            }
            else /* si el valor asignado a la etiqueta  */
            {    /* del nodo i no es cero                */
                tomar para el grupo par el valor
                            de la etiqueta asignada;
                tomar para el grupo impar el valor
                            de la etiqueta asignada;
            }
        }
        else /* i es par */
        {
            if(si la etiqueta impar asignada es cero)
            {
                para el grupo impar asignar 0.5;
                para el grupo par asignar 0.0;
            }
            else /* si la etiqueta asignada no es cero */
            {
                para el grupo impar asignar la etiqueta de i;
                para el grupo par asignar la etiqueta de i;
            }
        }
    } /* fin de for */
/* para el grupo impar revisar las etiquetas asignadas por el
   valor de 0.5 para cambiar a uno o a cero                  */
    for(i=1; i<=número de nodos; i++)
    {
        if(si el grupo impar i es igual a 0.5)
        {
```

```
            for(j=1; j<=número de nodos adyacentes a i; j++)
            {
                if(si algún nodo adyacente a i
                        es mayor  que  0.5)
                {
                    cambiar el grupo impar de 0.5 a 1 en i;
                    break; /* pasar a i+1 */
                }
                else if(si paso por todos los nodos adyacentes sin
                         encontrar un valor mayor a 0.5)
                {
                    para el grupo impar de i asignar
                    el valor de cero;
                    impare++; /* número de ceros
                                 del grupo impar     */
                }
            }
        }
                                /* para el grupo par: */
        if(si en el grupo par i es igual a 0.5)
        {
            for(j=1; j<=número de nodos adyacentes a i; j++)
            {
                if(si hay algún nodo adyacente a i que sea
                   mayor de 0.5)
                {
                    asignar al grupo par i el valor de 1.0
                    break; /* continuar con i+1 */
                }
                else if(si no encontro un valor mayor a 0.5
                        en los nodos adyacentes)
                {
                    asignar al grupo par i el valor de cero;
                    pare++; /* número de ceros del grupo par */
                }
            }
        }
    }/* fin de for i */
    if(si el número de ceros es mayor en el grupo par)
    {
        tomar al grupo par como extensión admisible;
    }
    else
    {
        tomar al grupo impar como extensión admisible;
    }
} /* fin de extensión_admisible */
```

Subdivide_paralelo().- Una vez que las etiquetas asignadas a cada vértice de las subregiones son compatibles, se inicia la

subdivisión de cada región en forma paralela, es decir se divide una subregión en cuatro o tres partes y no se volverá a subdividir hasta no hacerlo con toda las subregiones. De esta manera si la etiqueta máxima de la red cuadrilátera regular es tres, pasara por la serie de subregiones ese número de veces. El procedimiento *Subdivide_paralelo()* administra los procesos de subdivisión balanceada y no balanceada asimismo el fin del proceso de discretización.

```
subdivide_paralelo()
{   do
    {
        for(i=1; i<=número de elementos; i++)
        {
            inicializa al contador de número de ceros;
            for(j=1; j<=4; j++)
            {
                almacena al nodo del vértice;
                almacena la etiqueta del vértice;
                almacena coordenadas del vértice;
                if(etiq == 0)
                    cuenta el número de ceros;
                else
                {
                    vértice diferente de cero;
                    elemento en que se encuentra;
                }
            }
            if(número de ceros del elemento == 3)
                subdivide_1(vertice, elemento); /* hay una
                                                   etiqueta diferente
                                                   de cero     */
            else if(número de ceros == 4)
                contar++; /* el elemento ya no puede ser dividido
                             llevar su registro            */
            else
                subdivide_2(); /* hay dos o tres etiquetas
                                  diferentes de cero       */
        }/* fin de for i */
        if(contar es diferente al número de elementos)
        {
            inicializar variables para el nuevo nivel de
            división;
            cambiar nivel de almacenamiento;
        }
        else
        {
            el proceso de generar elementos cuadrangulares
            termino;
```

```
            exit;       /* salir del proceso */
        }
    }while(1) /* ciclo infinito */
}/* fin de subdivide_paralelo */
```

Subdivide_1().- Este algoritmo se encarga de la subdivisión no balanceada, como se explica en el inciso 3.5.5. Se aplica en las subregiones donde hay únicamente un vértice diferente de cero, figura 3.13.

```
subdivide_1()
{
    almacena la etiqueta diferente de cero;
    for(i=1; i<=4; i++)
    {
        nod[i] almacenamiento local de los nodos en proceso;
        coord[i][0] coordenada en x;
        coord[i][1] coordenade en y;
    }
    calcular las nuevas etiquetas;
    calcular las coordenadas de los puntos medios de cada borde;
    calcular las coordenadas del punto central;
    linea 1;    /* traza las lineas que dividen al elemento*/
    linea 2;
    linea 3;
    ampliar memoria dinámica;
    almacenar los valores calculados;
    aumentar el numero de elementos globales en tres;
} /* fin de subdivide_1 */
```

Subdivide_2().- Este algoritmo se encarga de la subdivisión balanceada, como se explica en el inciso 3.5.4. Se aplica en las subregiones donde por lo menos dos vértices no adyacentes, son diferentes de cero, figura 3.12.

```
subdivide_2()
{
    for(i=1; i<=4; i++)
    {
        calcular etiquetas de esquinas;
    }
    for(i=1; i<=4; i++)
    {
        calcular etiquetas de puntos medios;
    }
```

```
    for(i=5; i<9; i++)
    {
        calcula etiqueta central;
    }
    for(i=1; i<=4; i++)
    {
        calcula la coordenada de los puntos medios de cada borde;
    }
    for(i=5; i<7; i++)
    {
        calcula coordenadas centrales;
    }
    for(k=5; k<9; k++)
    {
        linea; /* lineas que dividen al elemento */
    }
    ampliar memoria dinámica;
    almacenar los datos calculados;
    numero_rostros+=4 /* aumento en elementos globales */
}/* fin de subdivide_2 */
```

Los procedimentos siguientes pertenecen al módulo que genera las mallas con elementos triangulares y corresponden al inciso 3.6.

Leer().- En este procedimiento se introducen los datos necesarios para el proceso de discretización con elementos triangulares. Los primeros datos son el número de nodos iniciales y el de huecos. Con esto se iniciliza la memoria dinámica para almacenar los bordes y coordenadas que se introducirán. También se determinan las coordenadas máximas y mínimas de la región que se discretizará para su visualización gráfica en la pantalla. Con el número de huecos se aparta memoria para los conectores parciales y las cadenas.

```
leer()
{   número de nodos iniciales;
    número de claros;
    crear memoria dinámica;
        para bordes;
        para coordenadas;
    for(i=1; i<=número de bordes; i++)
    {
        extremo(i,1)   /* Nodo extremo uno */
```

```
        extremo(i,2)    /* Nodo extremo dos */
        termino(i,1)    /* división del extremo 1 */
        termino(i,2)    /* división del extremo 2 */
    }
    for(i=1; i<=número de nodos; i++)
    {
        x(i);
        y(i);
        determinar xmin, xmax, ymin, ymax; /* rango de la
                                              pantalla     */
    }
    Crear espacios para conectores;
    if(número de huecos != de cero)
    {
        crear espacio en el puntero global;
        crear espacio para los conectores parciales superiores;
        crear espacio para el conjunto de conectores parciales
                                                    (cadenas);
    }
    for(i=1; i<=número de huecos; i++)
        nodo que activa al claro i-esimo;
}/* fin de función leer() */
```

Progresion().- Este procedimiento se basa en la expresión 3.7 del inciso 3.6.1. Su función es calcular el número de nodos y su ubicación en un borde o conector, según tamaño de elemento en cada uno de los extremos y la longitud del borde o conector.

```
void progresion(nodo_uno, nodo_dos, dis_dos)
{   x1; /* coordenadas del punto uno */
    y1;
    x2; /* coordenadas del punto dos */
    y2;
    long1; /* longitud del punto uno al dos */
    revisar condiciones;
    calcular proporción de la progresión;
    calcular el número de divisiones;
    calcular los coeficientes;
    calcular las coordenadas;

    /* almacenar coordenadas */

    incrementar el número de nodos globales;
    crear espacio para el nodo nuevo;
    crear espacio para coordenadas;
    return; /* regresar al lugar donde se llamo a progresion() */
} /* fin de la función progresion() */
```

Cond_inic()._ Una vez que se han introducido los datos de la región, es necesario establecer el número de nodos y su distribución en cada borde de la frontera los cuales activan la región que se va a discretizar; esto se realiza por medio del procedimiento *progresion()*.

```
void cond_inic()  /* condiciones iniciales */
{
    for(i=1; i<=número de bordes; i++)
    {
        /* recuperar informacion de los arreglos globales */
        n1;  /* nodo uno */
        n2;  /* nodo dos */
        dx1; /* division nodo uno */
        dx2; /* division nodo dos */
        /* se calculan los nodos de cada segmento por medio
          de una progresión geométrica */
        progresion(n1, n2, dx1, dx2);
        linea(n1, n2) /* trazo del borde en
                         el dispositivo gráfico */
    }
} /* fin de función cond_inic() */
```

Preproceso().- Este procedimiento se emplea cuando en la región hay huecos, su finalidad es crear las variables necesarias para su utilización posterior.

```
void preproceso()  /* esta rutina se utiliza sólo si hay huecos */
{    if(si el número de huecos es diferente de cero)
    {
        localizar espacio para nodos máximos;
        localizar espacio para nodos mínimos;
        localizar espacio para ordenar a los nodos máximos;
        localiza() /* busca a los nodos máximo y mínimo
                      de cada claro y ordena el arreglo de
                      nodos máximos de mayor a menor.        */
        /* construir conectores parciales superiores */
        for(i=1; i<= número de huecos; i++)
        {
            nodo1 = nodo máximo del hueco;
            for(j=0; j<=numero de huecos; j++)
                visibles(nodo1); /* calcula los nodos visibles para
                                    el nodo máximo nodo1.        */
```

```
            for(j=1; j<= número de nodos visibles; j++)
            {
                tomar el nodo visible con
                            la distancia más corta (nodo_co);
                tomar como segunda opción al nodo con el
                            ángulo mínimo máximo (nodo_co2);
            }
            clave = determinar(nodo_co); /* determinar a que
                                región pertenece dicho nodo */
            if(clave == 3 o a cero)
            {
                /* no existe o es erroneo dicho nodo tomar la
                                segunda opción (nodo_co2) */
                nodo_co = nodo_co2;
                clave = determinar(nodo_co);
                if(clave == 3 o es igual a cero)
                    ERROR;
            }
            if(clave == 1)
            {
                /* pertenece a la frontera */
                n_cadenas++; /* incremento en
                                el número de  cadenas. */
                CADNA(n_cadenas, 1) = i; /* conector parcial */
            }
            else if(clave == 2) /* pertenece a una cadena
                                                ya  creada. */
            {
                for(j=1; j<=número de cadenas; j++)
                    if(nodo_co == MIN(j))
                    {
                        buscar el número de cadena
                                a la que pertenece;
                        incremento en el número de conectores;
                    }
            }
            for(j=1; j<= número de huecos; j++)
                buscar el número del hueco;
            /* almacenar los datos del conector parcial */
            COPAR(i,1) = nodo del conector;
            COPAR(i,2) = nodo adyacente al nodo_conector;
            COPAR(i,3) = nodo del extremo opuesto;
            COPAR(i,4) = nodo adyacente al extremo opuesto;
            COPAR(i,5) = número de hueco;
        } /* fin de for i */
    } /* fin de if */
} /* fin de función preproceso */
```

Proceso().- El objetivo de este procedimiento es controlar el proceso de división de regiones por no ser convexas o tener algún

hueco.

```
void proceso()
{    if(número de huecos es diferente de cero)
        descomponer(); /* descomponer la región en subregiones
                                                simples.        */
     else
        dividir(); /* revisar que la región sea convexa */
     refinar(); /* suavizado final de la malla */

} /* fin de función proceso */
```

Descomponer().- Una vez que se ha determinado que en la región existen huecos se emplea el procedimiento *descomponer()*, para su desarrollo se utilizó el inciso 3.6.3.

```
void descomponer(nodo activo)
{
    Revisar si en la región hay un hueco;
    Buscar los límites de la región;
    Comparar si los nodos del hueco se encuentran
                                fuera de la región;
    if(número de huecos es diferente de cero)
    {
        Tomar el nodo máximo del hueco;
        Buscar el menor de los máximos;
        Buscar los nodos visibles para este nodo menor;
        Incrementar el número de conectores parciales;
        Tomar de los nodos visibles el de menor distancia;
        Buscar la cadena en donde almacenar el nuevo conector
                                                  parcial;
        Insertar en el conector parcial nodos de acuerdo a la
              distancia promedio de los nodos de los extremos;
        Revisar recursivamente si hay otro hueco en la región
                                                  resultante;
    }
    else
        dividir(); /* revisar si la región es convexa */
    if(número de conectores == 0)
        reordenar la cadena en dirección reversiva;
    descomponer() /* Revisar si la nueva subregión tiene algún
                                                  hueco;
} /* fin de la función descomponer */
```

Dividir().- Cuando la región no es convexa o las condiciones para

crear elementos son inadecuadas se emplea el procedimiento *dividir()*. Este procedimiento se basa en el inciso 3.6.2 del capitulo anterior.

```
void dividir()
{
    revisar si la región es convexa;
    if(si fue convexa)
    {
        crear una nueva subregión activa dividiendo
                                la región no convexa;
        dividir(); /* comprobar que la región resultante
                    sea convexa llamando recursivamente
                    a la función dividir()                  */
    } /* fin de if */
    else
        mallar(); */ si es convexa
                     triangular dicha región */
} /* fin de la función dividir() */
```

Mallar().- Cuando la región es convexa,el procedimiento *mallar()* es empleado para su discretización. Para el desarrollo de este procedimiento se utilizó el inciso 3.6.4. También dentro de *mallar()* se llaman a los procedimientos *cerrar()* y *bisectar()* de acuerdo al valor del ángulo.

```
void mallar( nodo_activo )
{
    falso_n = transferencia(nodo_activo, -1)

    for(;;)
    {
        if( transferencia(nodo_activo, 3) == nodo_activo)
            break; /* si la región es un triángulo
                                termina el proceso */
        /* Calcular el ángulo que forma el nodo */
        an_interior = angulo(nodo_anterior, nodo_activo,
                                            nodo_posterior);
        if(an_interior <= ANG_MN)  /* ANG_MN = 85. */
        {
            /* cerrar el ángulo */
            cerrar(nodo_anterior, nodo_activo, nodo_posterior)
            clave1 = 1;
        }
        else if( an_interior <= ANG_MY)  /*ANG_MY = 125. */
        {
            /* insertar un nodo nuevo */
            bisectar( nodo_anterior, nodo_activo, nodo_posterior);
            if(clave1 != 0) /* clave1 es una variable global que
```

```
                        si su valor es diferente de cero el
                        proceso de bisectar se realizó con
                        éxito.                          */
            if(nodo_activo == falso_n)
            {
                /* se recorrieron todos los nodos sin
                   encontrar un nodo apropiado para
                   el proceso bisectar                  */
                dividir2(falso_n, 0); /* dividir la región */
                return; /mallar las subregiones resultantes */
            }
        }
        else  /* no encontró un ángulo adecuado */
        {
            clave1 = 0;
            if(falso_n == nodo_activo)
            {
                /* se recorrieron todos los nodos sin  ecntrar
                           sin encontrar un ángulo adecuado */
                dividir2(falso_n, 0);
                return;
            }
        }
        if(clave1 != 0)
         /* se llevo a cabo con exito cerrar() o bisectar()
             cambiar de nodo activo                     */
            falso_n = transferencia(nodo_activo, -1);
        nodo_activo = transferencia(nodo_activo, 1);
    } /* fin de for infinito */
} /* fin de función mallar() */
```

Cerrar().- Este procedimiento es empleado cuando en el ángulo que forman dos nodos activos no es posible insertar un nodo, inciso 3.6.4.

```
void cerrar(n0, n1, n2)
{    almacen(n0, n1, n2); /* inicializa y almacena a los nodos
                             n0, n1 y n2 y al elemento que
                                                  forman */
    alterar(n0, n2); /* adiciona la lista nodo-nodo adyacente */
    linea(n0, n2); */ trazo gráfico de línea */
    /* revisa si es posible realizar el suavizado parcial
                                              de la malla */
    checar(n0);
    checar(n2);
} /* fin de la función cerrar() */
```

Bisectar().- Si en el proceso de discretización con *mallar()* dos nodos activos forman un ángulo cercano a 120• se inserta un nodo nuevo, inciso 3.6.4. Este proceso se realiza con el procedimiento *bisectar()*.

```
void bisectar(nd0, nd1, nd2)
{    /* rotar 60•contra las manecillas del reloj el primer lado
        para obtener las coordenadas del triángulo equilátero
        como en la figura 3.24                                    */
     ptr_rt = rotar(x1, y1, x0, y0, 1);
     x0 = *ptr_rt;       /* coordenadas calculadas */
     y0 = *(ptr_rt+1);
     /* rotar 60• con el sentido de las manecillas del reloj el
        segundo lado para obtener las coordenadas de este       */
     ptr_rt = rotar(x1, y1, x2, y2, -1);
     x2 = *ptr_rt;       /*coordenadas del segundo lado */
     y2 = *(ptr_rt+1);
     /* calcula el punto medio de los dos pares anteriores */
     xm = (x2 - x0)/2 + x0;
     ym = (y2 - y0)/2 + y0;
     /* calcular los nodos visibles para nd1 con el fin de
        aceptar o rechazar el punto medio                       */
     vistos = visibles(nd1, transferencia(nd1, 1))
     aceptar = 0;
     /* revisa que los nodos no se traslapen */
     for(i=0; i<=nodos_visibles; i++)
     {
         aceptar++;
     }
     if(aceptar == nodos_visibles)
     {
         /* no se traslapo con ningún elemento
            aceptar dicho nodo.                    */
         nnodosg++; /* incremento en el número de nodos
                                                   globales */
         definir espacio para el nuevo nodo;
         definir espacio para coordenadas;
         alterar(nd1, nnodosg);  /* queda fuera de actividad nd1 */
         alterar(nnodosg, nd2);  /* activa nuevos nodos          */
         alterar(nd0, nnodosg);
         linea(nnodosg, nd0); /* traza en la pantalla los  */
         linea(nnodosg, nd1); /* nuevos elementos.         */
         linea(nnodosg, nd2);
         /* almacena los dos elementos creados      */
         almacen(nnodosg, nd0, nd1);
         almacen(nnodosg, nd1, nd2);
         checar(nnodosg); /* reliza refinamiento local */
         checar(nd0);
         checar(nd2);
         clave1 = 1 /* se realizó con éxito bisectar */
```

```
    }
    else
        clavel = 0; /* se traslapa con algún elemento */
} /* fin de función bisectar() */
```

Se han presentado los procedimientos principales que se implementaron con base en lo expuesto en el capitulo tres. En el siguiente capitulo se exponen las aplicaciones gráficas efectuadas con el *sistema gráfico de generación de mallas para el método del elemento finito.*

APLICACIONES

En este capitulo se presentan diferentes ejemplos de generación de mallas con elementos finitos triangulares y cuadrangulares utilizando los algoritmos que se describieron en el capitulo tres e implementados como se mencionó en el capitulo cuatro.

5.1) Generacion de mallas con elementos cuadrangulares.

Para aplicar el algoritmo que genera elementos cuadrangulares, se ha selecionado una región con frontera curva y con dos huecos en su interior. Además se puede refinar una subregión predeterminada y escoger diferentes niveles de refinamiento sin tener que comprobar la conformidad entre subregiones vecinas ya que esto es asegurado automáticamente por el algoritmo. En la figura 5.1 se muestran las condiciones iniciales de la región y la numeración asignada a cada subregión, para poder referirse a su ubicación. Los datos iniciales de la región (nodos de cada subregión y coordenadas de cada nodo), se listan en el anexo B

5.1.1) Ejemplo 1.

En la figura 5.2 se observan las condiciones finales de la región discretizada que corresponde a un nivel de subdivisión

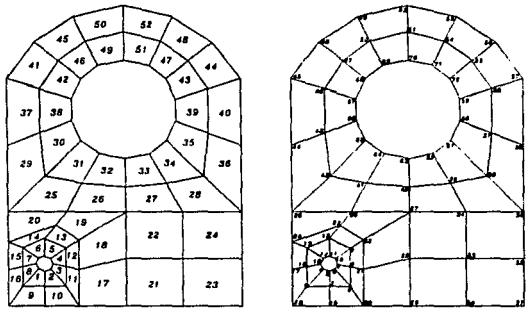


Fig 5 1 Condiciones iniciales

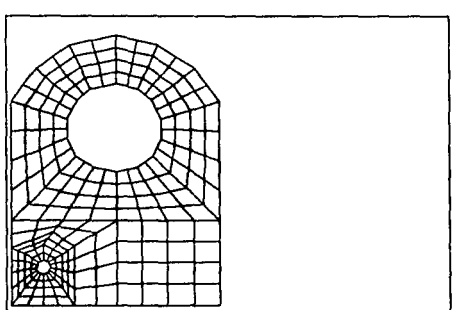


Fig.5.2 Discretización uniforme, ejemplo 1. Se generaron 208 elementos y 250 nodos.

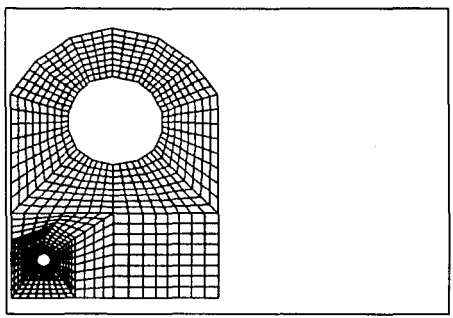


Fig.5.3 Discretización uniforme, ejemplo 1. Se generaron 916 nodos y 832 elementos.

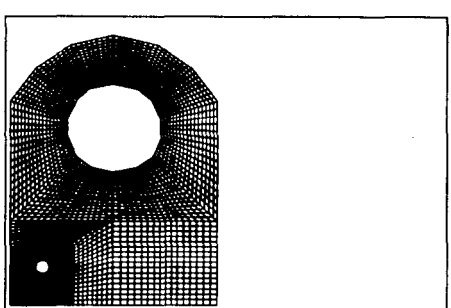


Fig.5.4 Discretización uniforme, ejemplo 1. Se generaron 3,496 nodos y 3,328 elementos.

uniforme igual a 1 para todas las subregiones. Con este dato se generaron 208 elementos y 250 nodos. Para la siguiente demostración se asigna a todas las subregiones un nivel de subdivisión 2, obteniendo una malla uniforme pero con elementos de menor tamaño que los generados anteriormente, como se muestra en la figura 5.3. Con este dato se generaron 916 nodos y 832 elementos. Para la figura 5.4, se asigna a todas las subregiones un nivel de 3 para dividir cada subregión en un mínimo de 4^3 elementos. De esta manera se obtienen $4^3 \times 52 = 3,328$ elementos y 3,496 nodos.

5.1.2) Ejemplo 2.

Para este ejemplo, sólo a la subregión 40, figura 5.1, se le asigna un nivel de subdivisión de 3. Como se observa en la figura 5.5, se obtiene un refinamiento local de esta subregión. Con este único nivel de subdivisión se obtuvieron 178 elementos y 200 nodos, y se observa que la condición de conformidad entre subregiones vecinas (cambio suave en el tamaño de los elementos), está asegurada automáticamente.

Se asigna a la subregión 23, figura 5.1, un nivel de subdivisión de 4, obteniendo 467 elementos y 504 nodos. El resultado se observa en la figura 5.6.

Por último se asigna el valor de cuatro como único nivel de subdivisión de la subregión central 22, figura 5.1. Los resultados gráficos se observan en la figura 5.7. En éste se generaron 687 elementos y 726 nodos. Estas tres ultimas figuras, son una demostración de refinamiento aislado de una subregión.

A continuación, se seleccionan cuatro subregiones vecinas (49, 50, 51 y 52), y se les asigna el valor de tres como nivel de subdivisión. Los resultados se observan en la figura 5.8. Notese

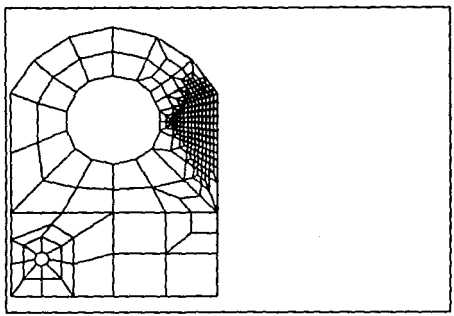


Fig.5.5 Refinamiento local, ejemplo 2. Se generaron 200 nodos y 178 elementos.

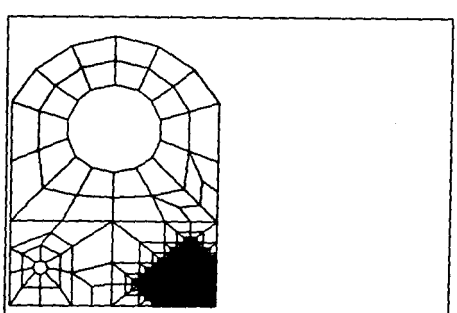


Fig.5.6 Refinamiento local, ejemplo 2. Se generaron 504 nodos y 467 elementos.

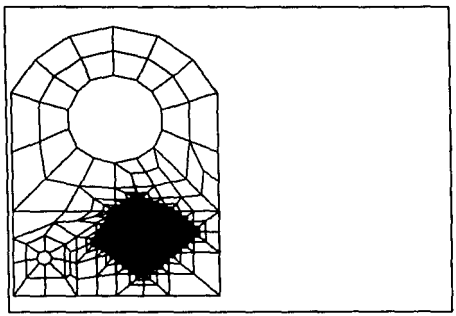


Fig.5.7 Refinamiento local, ejemplo 2. Se generaron 726 nodos y 687 elementos.

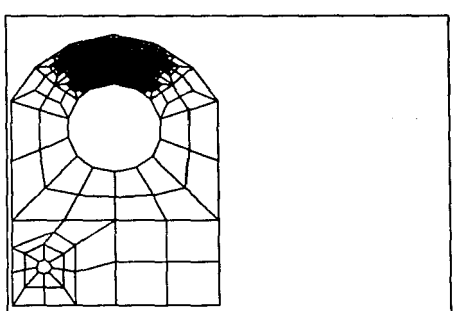


Fig.5.8 Refinamiento local, ejemplo 2. Se generaron 436 nodos y 400 elementos.

que la conformidad entre los bordes de las subregiones con nivel cero y cercanas al grupo, se asegura automáticamente.

5.1.3) Ejemplo 3.

Para el ejemplo tres, se efectúa una combinación de diferentes niveles de subdivisión, éstos son distribuidos como lo indica la tabla siguiente:

subregión	nivel	subregión	nivel	subregión	nivel
1	0	19	0	37	2
2	0	20	2	38	0
3	0	21	0	39	0
4	0	22	0	40	2
5	0	23	3	41	0
6	0	24	0	42	0
7	0	25	0	43	0
8	0	26	0	44	0
9	1	27	0	45	0
10	0	28	0	46	0
11	1	29	0	47	0
12	0	30	0	48	0
13	1	31	0	49	1
14	0	32	3	50	1
15	0	33	3	51	1
16	0	34	0	52	1
17	0	35	0	--	-
18	2	36	0	--	-

El dominio discretizado con los datos de la tabla anterior se muestran en la figura 5.9. Se generaron 393 elementos y 439 nodos. En este ejemplo se observa como se pueden combinar los niveles de subdivisión y su discretización uniforme; también se observa como influye sobre las regiones vecinas el refinado local.

5.1.4) Ejemplo 4.

En este ejemplo también se varian los niveles de división pero ahora alrededor del hueco mayor. La distribución de los

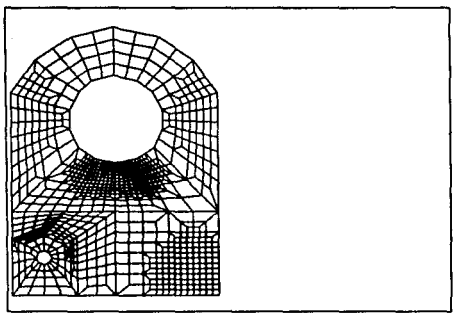


Fig.5.9 Combinación de diferentes niveles de subdivisión, ejemplo 3. Se generaron 439 nodos y 393 elementos.

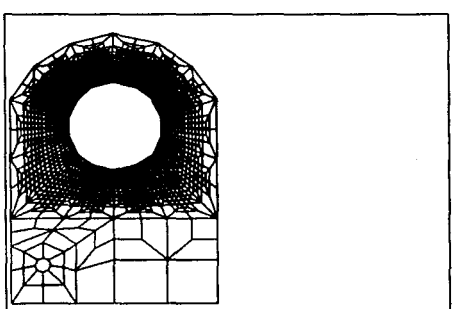


Fig.5.10 Refinamiento entorno de una región de interés, ejemplo 4. Se generaron 1,499 nodos y 1,400 elementos.

niveles de subdivisión de las subregiones se muestra en la tabla siguiente:

subregión	nivel	subregión	nivel	subregión	nivel
1	0	19	0	37	0
2	0	20	0	38	3
3	0	21	0	39	3
4	0	22	0	40	0
5	0	23	0	41	0
6	0	24	0	42	3
7	0	25	0	43	3
8	0	26	0	44	0
9	0	27	0	45	0
10	0	28	0	46	3
11	0	29	0	47	3
12	0	30	3	48	0
13	0	31	3	49	3
14	0	32	3	50	0
15	0	33	3	51	3
16	0	34	3	52	0
17	0	35	3	--	-
18	0	36	0	--	-

Con estos niveles se logra un refinamiento en las subregiones donde se espera que la variación de esfuerzos sean altos. Por ejemplo, en este caso la zona de interes es el hueco mayor, figura 5.10. Con los datos de la tabla anterior se generaron 1,400 elementos y 1,499 nodos.

5.1.5) Ejemplo 5.

En el ejemplo cinco, se utiliza tanto la generación de elementos uniformes como el refinamiento mayor de zonas particulares. Los datos de entrada se muestran en la tabla siguiente:

subregión	nivel	subregión	nivel	subregión	nivel
1	3	19	2	37	0
2	2	20	3	38	2
3	0	21	0	39	0
4	2	22	0	40	2
5	3	23	0	41	2
6	3	24	0	42	0
7	0	25	0	43	2
8	2	26	3	44	0
9	2	27	0	45	0
10	0	28	2	46	3
11	2	29	2	47	3
12	0	30	3	48	2
13	0	31	0	49	0
14	0	32	3	50	2
15	2	33	2	51	2
16	0	34	0	52	0
17	1	35	3	--	-
18	1	36	0	--	-

Los resultados gráficos se muestran en la figura 5.11. Con estos datos se generaron 1,501 elemento y 1,633 nodos.

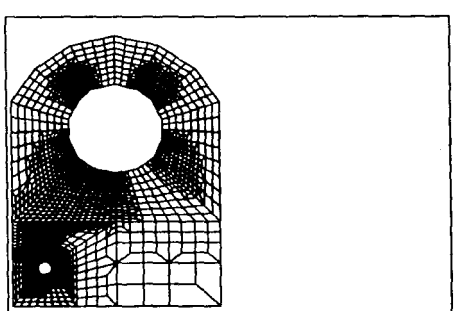


Fig.5.11 Discretización uniforme y refinamiento, ejemplo 5. Se generaron 1,633 nodos y 1,501 elementos.

5.2) Generación de mallas con elementos triangulares.

Para la generación de mallas con elementos triangulares se estudian cinco tipos de regiones; la primera es convexa y fácil de dircretizar; en ésta se varía el tamaño de los elementos así como su concentración. La segunda región es no convexa por lo que como primer paso se dividen en una forma más adecuada para su discretización. La tercera región tiene en su interior una abertura siendo necesario utilizar el procedimiento *descomponer* del algoritmo descrito en el inciso 3.6.3, con el cual se obtienen subregiones adecuadas para la discretización. La cuarta región es una forma no convexa complicada, con vértices agudos y en la que se varían las concentraciones de los elementos. La quinta región y última de la serie es un círculo con el cual se comprueba la facilidad del elemento triangular para describir fronteras curvas.

5.2.1) Ejemplo 1.

Para el primer ejemplo, se varió el tamaño de los elementos, conservando la forma de la región, en la figura 5.12 se muestra dicha región. Al iniciar la generación de elementos triangulares el primer dato es el *número de nodos iniciales* que corresponde al número de vértices de la frontera. De esta manera, por cada nodo inicial hay un borde. Por tal razón, el número de nodos iniciales corresponde al número de bordes necesarios n y el siguiente dato sera: *borde* n *nodo1?*, *nodo2?*, *d1?*, *d2?*. Por cada borde n se introduce el nodo de cada extremo (*nodo1*, *nodo2*), así como el tamaño deseado en cada extremo (*d1*, *d2*); los últimos datos son las coordenadas de cada nodo inicial.

Para el ejemplo de la figura 5.12 los datos que se introducen y su orden son los siguientes:

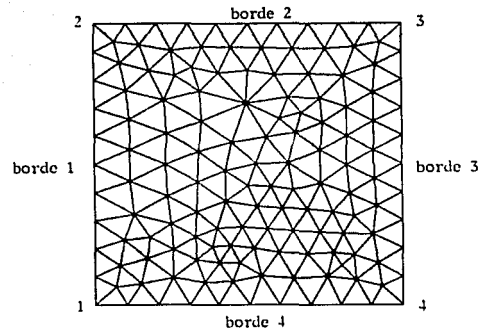


Fig.5.12 Discretización de una región convexa, ejemplo 1. Se generaron 142 nodos y 242 elementos.

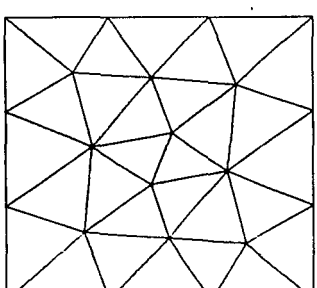


Fig.5.13 Discretización de una región convexa, ejemplo 1. Se generaron 22 nodos y 30 elementos.

número de nodos iniciales 4

borde	nodo1	nodo2	d1	d2
1	1	2	0.10	0.10
2	2	3	0.10	0.10
3	3	4	0.10	0.10
4	4	1	0.10	0.10

nodo	x	y
1	1.00	1.00
2	1.00	2.00
3	2.00	2.00
4	2.00	1.00

Como se puede observar en la figura 5.12, la numeración considerada en los nodos que describen inicialmente la frontera, es en el sentido de las manecillas del reloj, esta condición debe tomarse en cuenta ya que automáticamente se considera como región a discretizar aquella que se encuentra a la derecha del sentido en que se describió inicialmente el orden de los nodos.

En este ejemplo se asignó a los elementos un tamaño de 0.1 de manera uniforme en todos los extremos. Tomando en cuenta que todos los bordes tienen una longitud unitaria el número de elementos que se generaron en cada uno es igual a diez, como se muestra en la figura 5.12.

Con este tamaño de elemento se generaron 142 nodos y 242 elementos, describiendo unicamente una región inicial con cuatro nodos. Aunque esta es una región convexa, se efectuó el procedimiento de subdivisión para mejorar la distribución de los elementos y una transición de tamaño más suave (conformado).

En la figura 5.13 se muestra la misma figura, pero ahora el tamaño de los elementos es como se muestra en la siguiente tabla:

borde	nodo1	nodo2	d1	d2
1	1	2	0.30	0.30
2	2	3	0.30	0.30
3	3	4	0.30	0.30
4	4	1	0.30	0.30

El número de elementos generados es menor ya que se escogió un tamaño mayor. El tamaño de los elementos resulto aproximado a 0.30, esto es debido a que se utiliza una progresión geométrica, definida en el inciso 3.6.1, que calcula el tamaño final del elemento. Como la longitud del borde es unitaria se generan tres elementos por lo que el tamaño final de los elementos es de 0.3333. Para ubicar a los nodos intermedios la progresión toma en cuenta la longitud del borde y tamaño de cada elemento y determina el número de elementos que deben generarse en el borde. Con estos datos se generaron 30 elementos y 22 nodos, figura 5.13.

Ahora se concentrará un número mayor de elementos en la parte superior de la región, por consiguiente debemos especificar un tamaño de elemento menor en los nodos de los vértices superiores; nuestros datos de entrada en este caso serán:

borde	nodo1	nodo2	d1	d2
1	1	2	0.10	0.04
2	2	3	0.04	0.04
3	3	4	0.04	0.10
4	4	1	0.10	0.10

Con este tamaño de elemento se generaron veinticinco elementos en el borde dos, ver figura 5.14. Como se puede apreciar, la concentración de nodos se encuentra en este borde y la menor en el opuesto. Con esta concentración se generaron 428 nodos y 789 elementos. Además se observa que indirectamente se genera un refinamiento mayor en una parte de la frontera.

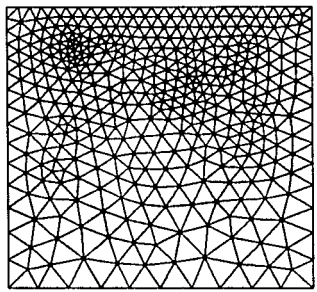


Fig.5.14 Discretización de una región convexa, ejemplo 1. Se generaron 428 nodos y 789 elementos.

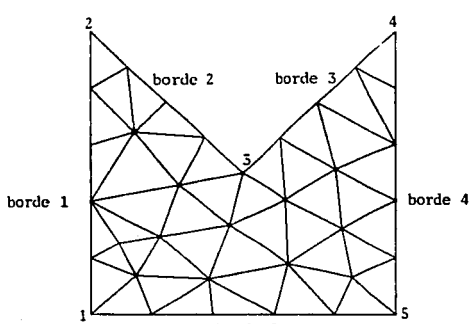


Fig.5.15 Discretización de una región no convexa, ejemplo 2. Se generaron 35 nodos y 45 elementos.

5.2.2) Ejemplo 2.

En el ejemplo 2 de la figura 5.15, se muestra una región no convexa sencilla la cual se dircretiza dandole un tamaño uniforme a los elementos con los siguientes datos:

borde	nodo1	nodo2	d1	d2
1	1	2	0.20	0.20
2	2	3	0.20	0.20
3	3	4	0.20	0.20
4	4	5	0.20	0.20
5	5	1	0.20	0.20

Coordenadas

nodo	x	y
1	1.00	1.00
2	1.00	2.00
3	1.50	1.50
4	2.00	2.00
5	2.00	1.00

En este caso como la región es no convexa primero se divididie en dos partes siguiendo el criterio descrito en el capitulo tres: tomar el ángulo interior mayor para dividir la región. Con este criterio, se toma al nodo 3 el cual se une a un nodo del borde 5, el más cercano al nodo 3; en seguida, se revisa la región de la izquierda , ya que el primer nodo del conector es el 3, como esta subregión es convexa se comienza a dircretizar. Después de triangular la subregión, se reordena el conector que divide a la región induciendo a ser la nueva subregión activa la parte derecha de la región. Con estos datos se generaron 35 nodos y 45 elementos.

En esta misma región se pueden crear elementos más regulares, definiendo un tamaño menor de elementos para ello se introducen los siguientes datos:

borde	nodo1	nodo2	d1	d2
1	1	2	0.10	0.10
2	2	3	0.10	0.10
3	3	4	0.10	0.10
4	4	5	0.10	0.10
5	5	1	0.10	0.10

Como se puede apreciar en la figura 5.16 al reducir el tamaño del elemento, este tiende a mejorar su forma aproximandose más a la de un triángulo equilátero; con estos datos se generaron 170 elementos y 108 nodos.

Para mejorar aún más la forma de los elementos de manera que se aproxime a la de un triángulo equilátero, se redujo más el tamaño de los elementos, como se muestra en la figura 5.17, los datos que se introdujeron son:

borde	nodo1	nodo2	d1	d2
1	1	2	0.05	0.05
2	2	3	0.05	0.05
3	3	4	0.05	0.05
4	4	5	0.05	0.05
5	5	1	0.05	0.05

con estos datos se generaron 770 elementos y 430 nodos y como se observa la calidad del elemento ha mejorado. Es necesario obtener elementos con forma regular para lograr una aproximación adecuada del MEF, pero no se debe descuidar la capacidad de memoria que puede ser limitante al tener muchos elementos.

5.2.3) Ejemplo 3.

Hasta ahora las regiones discretizadas han sido simples, en el siguiente ejemplo se muestra una región con un hueco. En este ejemplo un nuevo dato es el *Nodo activo del hueco*. Como se explicó en el capitulo tres, ese nodo es necesario para activar los nodos

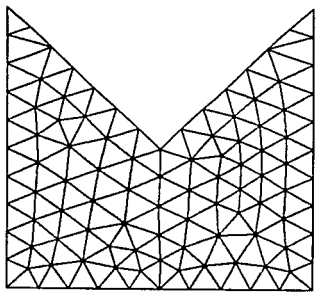


Fig.5.16 Discretización de una región no convexa, ejemplo 2. Se generaron 108 nodos y 170 elementos.

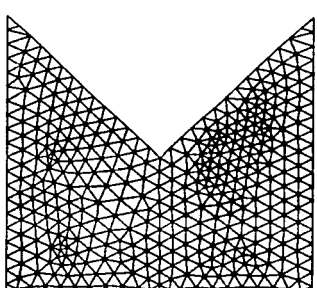


Fig.5.17 Discretización de una región no convexa, ejemplo 2. Se generaron 430 nodos y 770 elementos.

de la frontera del hueco y evitar que se queden aislados de los nodos de la frontera exterior. Otro dato nuevo es el *Número de huecos*, el cual define el número de aberturas en la región. Con estas nuevas variables los datos para este ejemplo son:

Número de nodos iniciales 8
Número de huecos 1
bordes:

borde	nodo1	nodo2	d1	d2
1	1	2	0.05	0.05
2	2	3	0.05	0.05
3	3	4	0.05	0.05
4	4	1	0.05	0.05
5	5	6	0.05	0.05
6	6	7	0.05	0.05
7	7	8	0.05	0.05
8	8	5	0.05	0.05

Coordenadas:

nodo	x	y
1	1.00	1.00
2	1.00	2.00
3	2.00	2.00
4	2.00	1.00
5	1.50	1.20
6	1.80	1.50
7	1.50	1.80
8	1.20	1.50

Nodo activo del hueco 5

Los resultados se pueden observar en la figura 5.18. Se generaron 784 elementos y 450 nodos.

5.2.4) Ejemplo 4.

En uno de los primeros ejemplos, figura 5.15, se mostro una

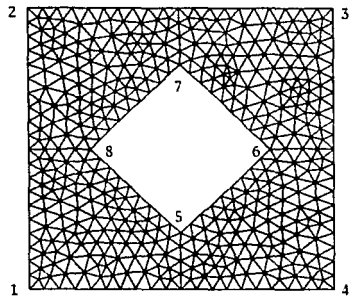


Fig.5.18 Discretización de una región compuesta, ejemplo 3. Se generaron 450 nodos y 784 elementos.

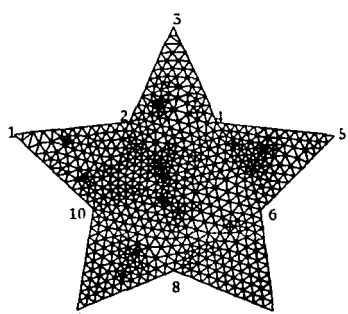


Fig.5.19 Discretización de una región no convexa complicada, ejemplo 4. Se generaron 650 nodos y 1,186 elementos.

región no convexa la cual se dividió formando subregiones más simples para su discretización. Ahora se muestra como triangular una región no convexa complicada como la figura 5.19. Los datos de entrada son:

borde	nodo1	nodo2	d1	d2
1	1	2	0.08	0.08
2	2	3	0.08	0.08
3	3	4	0.08	0.08
4	4	5	0.08	0.08
5	5	6	0.08	0.08
6	6	7	0.08	0.08
7	7	8	0.08	0.08
8	8	9	0.08	0.08
9	9	10	0.08	0.08
10	10	1	0.08	0.08

Coordenadas:

nodo	x	y
1	0.08	1.31
2	0.74	1.39
3	1.00	2.00
4	1.28	1.38
5	1.88	1.31
6	1.82	0.83
7	1.89	0.18
8	1.00	0.44
9	0.42	0.18
10	0.52	0.83

En este caso se tomo como tamaño aproximado del elemento 0.08 en todos los extremos de los bordes. Con estos datos se generaron 1,188 elementos y 650 nodos. Como lo muestra la figura 5.19, es posible dircretizar fronteras complicadas y regiones no convexas tan solo al especificar el contorno de la región.

Con los siguientes datos se concentra un número mayor de elementos en las puntas agudas de la figura 5.19, para ello en estas puntas se asigna un tamaño menor, por consiguiente los nuevos datos para los bordes son:

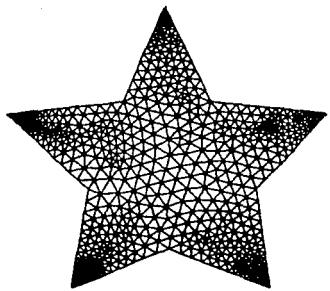


Fig.5.20 Concentración de elementos, ejemplo 4. Se generaron 632 nodos y 1,111 elementos.

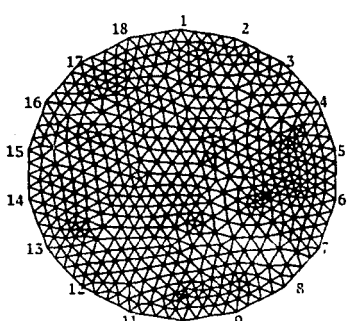


Fig.5.21 Discretización de una región con fronteras curvas, ejemplo 5. Se generaron 592 nodos y 1,110 elementos.

borde	nodo1	nodo2	d1	d2
1	1	2	0.02	0.08
2	2	3	0.08	0.02
3	3	4	0.02	0.08
4	4	5	0.08	0.02
5	5	6	0.02	0.08
6	6	7	0.08	0.02
7	7	8	0.02	0.08
8	8	9	0.08	0.02
9	9	10	0.02	0.08
10	10	1	0.08	0.02

Los resultados se observan en la figura 5.20, donde se aprecia la formación de un mayor refinamiento cerca de los vértices impares. De esta manera se generaron 1,111 elementos y 632 nodos.

5.2.5) Ejemplo 5.

Otro problema que se presenta en un generador de mallas es el tratar regiones con fronteras curvas. En el siguiente ejemplo se maneja una región circular mostrando la forma de los elementos triangulares y su distribución en la frontera con los siguientes datos:

bordes:

borde	nodo1	nodo2	d1	d2
1	1	2	0.08	0.08
2	2	3	0.08	0.08
3	3	4	0.08	0.08
4	4	5	0.08	0.08
5	5	6	0.08	0.08
6	6	7	0.08	0.08
7	7	8	0.08	0.08
8	8	9	0.08	0.08
9	9	10	0.08	0.08
10	10	11	0.08	0.08
11	11	12	0.08	0.08
12	12	13	0.08	0.08
13	13	14	0.08	0.08
14	14	15	0.08	0.08

15	15	16	0.08	0.08
16	16	17	0.08	0.08
17	17	18	0.08	0.08
18	18	01	0.08	0.08

Coordenadas:

nodo	x	y
1	1.00	2.00
2	1.34	1.94
3	1.64	1.77
4	1.87	1.50
5	1.98	1.17
6	1.98	0.83
7	1.87	0.50
8	1.64	0.23
9	1.34	0.06
10	1.00	0.00
11	0.66	0.06
12	0.36	0.23
13	0.13	0.50
14	0.02	0.83
15	0.02	1.17
16	0.13	1.50
17	0.36	1.77
18	0.66	1.94

Como se puede apreciar en la figura 5.21, el algoritmo puede tratar una región circular adecuadamente, adaptandose los elementos triangulares a la forma de la frontera. Con los datos anteriores se generaron 1,110 elementos y 592 nodos.

CAPITULO 6

CONCLUSIONES

Se presentaron dos algoritmos de generación de mallas propias para la solución de problemas con el MEF y que fueron implementados para desarrollar un generador automático de mallas de elementos triangulares y cuadrangulares, el cual discretiza regiones en dos dimensiones. Los datos de entrada para generar cada tipo de elemento es diferente ya que se sigue un procedimiento especifico en cada uno. Para los elementos cuadrangulares se divide la región en subregiones convexas, y regulares, introduciendo nodos iniciales y coordenadas de éstos. En el caso de los elementos triangulares se definen únicamente los nodos de los vértices de la región, las coordenadas y el tamaño del elemento deseado en cada vértice. En los dos casos sólo se necesita un mínimo de datos iniciales para discretizar la región en estudio y con el refinamiento que el usuario necesite.

Cada tipo de elemento tiene sus ventajas e inconvenientes según el problema por resolver. Ambos pueden ser utilizados en fronteras circulares o en regiones con huecos, como se mostró en el capítulo cinco.

Con base en los ejemplos mostrados se pueden dar las conclusiones siguientes:

Para mallas con elementos cuadrangulares:

La malla burda inicial debe de tener en las subregiones una

forma regular y un nivel de subdivisión. El nivel de subdivisión se asocia a la asignación de etiquetas admisibles a los vértices de cada subregión.

Simultaneamente se dividen todas las subregiones de la malla burda, teniendo en cuenta la etiqueta admisible de sus vértices, en subcuadriláteros y asi sucesivamente hasta que todas las etiquetas de los subcuadriláteros son cero.

Los cambios en el nivel de subdivisión de una subregión sólo afectan la distribución de la malla en una pequeña área permitiendo un refinamiento local de la región.

Un problema que se presenta en los generadores de mallas al dividir una región en subregiones es el comprobar la conformidad de los bordes, esto es, que los mismos elementos generados en un borde compartido por dos subregiones tengan el mismo número y tamaño. En el generador desarrollado esto no representa dificultad ya que la conformidad está asegurada automáticamente.

Al generar los elementos cuadrangulares se pueden convertir en triangulares adicionando correctamente una diagonal a sus vértices.

Para mallas con elementos triangulares:

El generador de mallas con elementos triangulares se inicializa de tal manera que el usuario define una región con una secuencia de lineas rectas y el tamaño del elemento deseado. Para generar los nodos de la frontera de la región se emplea una progresión geométrica consiguiendo que los datos iniciales sean los menos posibles.

La subdivisión y triangulación de una región se efectúa automáticamente de manera recursiva con el algoritmo de división y

triangulación implementados en procedimientos de lenguaje C. Estos procedimientos pueden ser utilizados en regiones no convexas y multiplemente conectadas (con uno o varios huecos).

La técnica de descomponer las regiones complejas en una forma más simple no sólo ayuda en el proceso de triangulación sino que provoca que los elementos de la malla tengan una transición de tamaño suave y una distribución más uniforme, lo cual es deseable en el análisis por el MEF; una vez que se obtienen las regiones simples, éstas se triangulan.

El proceso de triangulación se lleva a cabo elemento por elemento logrando un control sobre la forma de cada elemento y la ausencia, prácticamente nula, de triángulos obtusos. Durante la triangulación se efectuó dos veces el proceso de suavizado aumentando la calidad de la malla.

Se desarrolló una estructura de datos tal que: se puede activar una región por un sólo nodo de ésta; la rapidez en el proceso de suavizado es alta, y se puede remover una región triangulada facilmente de las que no lo están.

Es necesario mejorar la calidad del elemento triangular ya que se encuentran nodos que son compartidos por cuatro vértices lo cual provoca que los ángulos interiores de los elementos sean muy grandes (obtusos) y además se encuentran algunos nodos compartidos por ocho elementos provocando la creación de ángulos agudos.

Adicionales:

Es necesario una máquina (PC) que tenga una memoria RAM de más de 640 K bytes para obtener refinamientos mayores o bién correr las rutinas en una máquina de mayor capacidad (por ejemplo, una estación de trabajo).

Se requiere un monitor con una alta resolución (640 x 480) o auxiliarse de puertos para apreciar el refinamiento de la malla. Integrar a las rutinas nuevas utilerías en C (tal como ratón, puertos y ventanas).

Mejorar la forma de introducir los datos en el generador. Esto se puede lograr con el desarrollo de un editor propio para el generador. Además es deseable almacenar los datos generados en el disco duro para posteriormente utilizarlos con algún programa comercial de elemento finito

El empleo del lenguaje de programación C fué de gran utilidad en el diseño de la estructura de datos. Además el gran número de funciones gráficas permitió el despliege de los datos generados en la pantalla y obtener una copia de la malla final en una impresora tanto de puntos como de laser.

ANEXO A

En el anexo A se presenta el listado del programa fuente que se implemento en base a los algoritmos expuestos en los capitulos tres y cuatro. Con este código se formo el *sistema gráfico de generación de mallas para el método del elemento finito.*

```
/* Implementacion de un algoritmo generador
   de elementos triangulares amal.c          */

#include <stdio.h>
#include <math.h>
#include <graphics.h>
#include <alloc.h>
#define PI        3.141592654
#define GRA       180/PI                    /* valor de un radián */
#define ANG_MY    125                       /* ángulo mayor para bisectar */
#define ANG_MN    85                        /* ángulo menor para cerrar */
#define GAMA      70                        /* ángulo de visión */
#define PX        640                       /* pixeles en x */
#define PY        480                       /* pixeles en y */
#define NNA(i)    ((nodos+(i)-1)->nnodosa) /* número de nodos adyacentes */
#define NEA(i)    ((nodos+(i)-1)->nelementos) /* numero de elementos adyacentes */
#define ELNA(i,j) ((lelemento+(i)-1)->nudo[(j)-1]) /* lista elemento-nodos */
#define NDNA(i,j) *((nodos+(i)-1)->nodosa+(j)-1) /*lista nodo-elemento adyacentes */
#define NDEA(i,j) *((nodos+(i)-1)->elementos+(j)-1) /* lista nodo-nodo adyacente*/
#define PN(i)     ((nodos+(i)-1)->ptr1) /* puntero de nodos adyacentes*/
#define PE(i)     ((nodos+(i)-1)->ptr2) /* puntero de elementos adyacentes */
#define COORD(i,j) *(coord_ptr+((i)-1)*2+(j)-1)     /* lista de coordenadas */
#define DEFM1(i)  (struct dos*)malloc((i)*sizeof(struct dos)) /* listas */
#define DEF11(i)  (realloc(nodos, ((i)*sizeof(struct dos)) ) )
#define DEFM2(i)  ((float*)malloc((i)*2*sizeof(float)))
#define DEF22(i)  (realloc(coord_ptr, ((i)*2*sizeof(float))))
#define DEFM3(i)  ((int*)malloc((i)*sizeof(int)))
#define DEF33(a,i) (realloc(a, ((i)*sizeof(int))))
#define DEF44(i)  (realloc(lelemento,((i)*sizeof(struct seis))))
#define COPAR(i,j) ((conec_parc+(i)-1)->s[(j)-1])  /* conector parcial */
#define CADNA(i,j) *((cadena+(i)-1)->co_ptr+(j)-1) /* conjunto de c.p */
#define CONEC(i,j) ((conector+(i)-1)->es[(j)-1])   /* conector */
#define NUCOP(i)  ((cadena+(i)-1)->nconec)  /* número de c.p. */
#define ERROR     error()
#define X(i)      ( ( (i) + ddx-minimox) * esc)     /* x en el dispositivo */
#define Y(i)      ( PY - (ddy + (i)-minimoy) * esc) /* y en el dispositivo */


          /* estructuras globales */
struct uno
{
    int   extremos[2];
    float terminos[2];
};

struct dos
{
    int nnodosa;    /* número de nodos adyacentes */
    int nelementos; /* número de elementos adyacentes */
    int *elementos; /* elementos adyacentes al nodo */
    int *nodosa;    /* nodos adyacentes al nodo de la estructura */
    int *ptr1;      /* puntero de nodos adyacentes */
    int *ptr2;      /* puntero de elementos adyacentes */
};

struct tres         /* conectores */
{
    int es[4];
};
```

```
struct cuatro       /* conectores parciales */
{
    int s[5];
};

struct cinco
{
    int nconec;
    int *co_ptr;
};

struct seis
{
    int nudo[3];
};
                            /* variables globales */
int nbordes;                /* número de bordes */
int nagujeros;              /* número de agujeros */
int nnodosg;                /* número de nodos globales */
int nelementosg;            /* número de elementos globales */
struct uno *borde;          /* puntero de lista de bordes */
struct dos *nodos;          /* puntero de lista de nodos */
float *coord_ptr;           /* puntero de coordenadas */
int *nodo_activo;           /* puntero de nodos activos de claros */
int *maxi,                  /* puntero de lista de nodos máximos de agujero */
    *mayo,                  /* puntero de lista de nodos ordenados de agujero */
    *mini;                  /* puntero de lista de nodos mínimos de agujero */
int activo;                 /* inicializador del puntero nodos activos */
int nodos_pro;              /* nodos generados por la progresión */
int nod_visibles;           /* nodos visibles para un punto */
float ang1, ang2;           /* ángulos de los lados de un vertice */
struct tres *conector;      /* puntero de conectores */
struct cuatro *conec_parc; /* puntero de conectores parciales */
struct cinco *cadena;       /* puntero de conjunto de conectores parciales */
struct seis *lelemento;     /* puntero de lista elemento nodos adyacentes */
float esc, dix, diy;        /* escala del dispositivo gráfico */
float minimax, minimay;     /* origen del puerto */
int *diferencia;            /* diferencia de direcciones */
int clavel = 1;             /* se lleva acabo el proceso */
int n_cadenas;              /* número de cadenas */
int global;                 /* almacena etiqueta de agujero o claro */
int parametro;              /* parámetro de suavizado de nodos */
int n_pri, n_seg;           /* nodos primero y último de progresión */
int n_co;                   /* número de conectores  */
int posx;             /* ** */
int posy;             /* posición de impresión */
char cla_ve;

   /* funciones globales */
void progresion(int, int, float, float);
void adyacentes(int, int);
float dist(int, int);
int *visibles(int, int);
void inic(int);
void descomponer(int, int);
void dividir(int, int);
void dividir2(int, int);
void mallar(int);
void bisectar(int, int, int);
void cerrar(int, int, int);
```

```
void error();
int transferencia(int, int);
float p_dis(int);
float ang_mx(int, int);
void reordenar1(int*);
int determinar(int);
void inic_elem(int);
void cambio(int, int);
float cuadrante(float, float);
void pool();
float angulo(int, int, int);
void almacen(int, int, int);
float *rotar(float, float, float, float, int);
void refinado(int, int);
void checar(int);
void linea(int,int);
void resumen();
void vision();
int  busqueda(float, float);

/* La siguiente funcion tiene como fin introducir los datos necesarios
   para crear los elementos triangulares.
   Primeramente introducir: el numero de bordes que definen la frontera
   del dominio,contado los bordes que definen el claro o claros si los hay.
*/

void leer(void)
  /*------*/
#define EXTRE(i,j) ( (borde+(i)-1)->extremos[(j)-1] ) /* lista de extremos */
#define TERMI(i,j) ( (borde+(i)-1)->terminos[(j)-1] ) /* lista de terminos */
{
    register int i;
    int parc, n_1, n_2;
    float dx1, dx2, x1, y1;
    float xmin=0, ymin=0, xmax=0, ymax=0;
    void inic_graph( float, float);

    pool();
    printf("\nPosición de impresion x,y: x(mm), y(mm)\n");
    scanf("%d %d", &posx, &posy);        /* posición de impresión */

    printf("\nIntroducir el numero de nodos iniciales\n");
    scanf("%d", &nnodosg);          /* lee el numero de nodos inicial */
    printf("Numero de agujeros\n");
    scanf("%d", &negujeros);        /* lee el numero de claros en el dominio */
    nbordes   = nnodosg;
     /* Crea espacio en el pool para introducir los demas datos */
    if(coreleft()-nbordes*sizeof(struct uno) < 0)    /* bordes */
        ERROR;  /* revisa que no se exceda la capacidad de memoria */
    borde = (struct uno*)malloc(nbordes*sizeof(struct uno));
    if(coreleft()-nnodosg*2*sizeof(float) < 0)       /* coordenadas */
        ERROR;                                  /* revisa capacidad de memoria */
    coord_ptr = DEFM2(nnodosg);       /* crea espacio en la memoria activa */
    if(!coord_ptr) ERROR;            /* comprueba que el puntero no sea nulo */
     /* introduce los nodos de los extremos de
    cada borde la division inicial y final  */
    for(i=1; i<=nbordes; i++)
    {
    printf("borde %d nodo1?, nodo2?, d1?, d2?\n", i);
    scanf("%d %d %f %f", &n_1, &n_2, &dx1, &dx2);
```

```
/*  printf(" %d %d %f %f \n", n_1, n_2, dx1, dx2); */
    EXTRE(i,1) = n_1;       /* nodo del extremo 1 */
    EXTRE(i,2) = n_2;       /* nodo del extremo 2 */
    TERMI(i,1) = dx1;       /* division del extremo 1 */
    TERMI(i,2) = dx2;       /* division del extremo 2 */
    }
     /* introducir las coordenadas de los nodos iniciales o vertices */
    for(i=1; i<=nnodosg; i++)
    {
    printf("Coordenadas del nodo %d?\n", i);
    scanf("%f %f", &x1, &y1);     .
/*  printf(" %f %f \n", x1, y1);  */
    COORD(i,1) = x1;    /* abscisa */
    COORD(i,2) = y1;            /* ordenada */
    if(xmin == 0) xmin = x1;  /* Determina los limites */
    if(ymin == 0) ymin = y1;  /* de la pantalla  de    */
    if(xmin > x1) xmin = x1;  /* acuerdo con el        */
    if(xmax < x1) xmax = x1;  /* dominio a procesar    */
    if(ymin > y1) ymin = y1;
    if(ymax < y1) ymax = y1;
    }
    minimox = xmin;
    minimoy = ymin;
    dx1 = xmax - xmin;
    dx2 = ymax - ymin;
     /* crea espacio disponible para conectores */
    nodos =DEFM1(nnodosg); /* busca espacio disponible */
    if(!nodos) ERROR; /* no hay memoria disponible */
    for(i=1; i<=nnodosg; i++)
    {
    NNA(i) = 0; /* inicializa */
    NEA(i) = 0;
    }
    conector = (struct tres*)malloc(15*sizeof(struct tres));
    if(!conector) ERROR; /* no hay espacio disponible */
    if(nagujeros)       /* Si el numero de agujeros es diferente de cero... */
    {
    nodo_activo = DEFM3(nagujeros*1);   /* crea espacio para el puntero global */
        if(!nodo_activo)   ERROR;  /* si el puntero es nulo...salir   */
    *nodo_activo = 1; /* inicializacion de nodos activos */
    /* crea espacio para los conectores parciales superiores */
    conec_parc = (struct cuatro*)malloc(nagujeros*sizeof(struct cuatro));
    if(!conec_parc)
        ERROR; /* si el puntero es nulo no hay espacio disponible */
    /* crea espacio para el conjunto de conectores parciales(cadenas) */
    cadena = (struct cinco*)malloc(nagujeros*sizeof(struct cinco));
    if(!cadena) ERROR; /* no hay memoria disponible */
    for(i=1; i<=nagujeros; i++)
    {
        (cadena+i)->co_ptr = DEFM3(1);
        if(!(cadena+i)->co_ptr) ERROR;
        NUCOP(i) = 1;
    }
    } /* fin de if */
    for(i=1; i<=nagujeros; i++)
    {
    printf("Nodo activo del agujero %d?\n ", i);
    scanf("%d", &n_1);
    *(nodo_activo+i) = n_1;
    }
```

```
    /*      inicializa la pantalla con las caracteristicas de esta  */
    inic_graph(dx1, dx2);
    setbkcolor(LIGHTRED);  /* color fondo de la pantalla */

    /* inicializa el puntero de la lista de elementos-nodos */
    lelemento = (struct sels*)malloc(2*sizeof(struct sels));
 /*   vision(); */
}/* fin de leer() */
#undef TERMI
#undef EXTRE

/* La funcion cond_inic() genera los datos necesarios para
   crear los elementos triangulares                              */
void cond_inic()
  /*------------*/
#define EXTRE(i,j)  ((borde+(i)-1)->extremos[(j)-1]) /* lista de extremos */
#define TERMI(i,j)  ((borde+(i)-1)->terminos[(j)-1]) /* lista de terminos */
{
    int   n_1, n_2;
    float dx1, dx2;
    int *ndl;     /* apuntador de nuevos nodos */
    int alma;
    register int i,j;          /* variables for */

    for(i=1; i<=nbordes; i++)
    {
    n_1 = EXTRE(i,1);                /* nodo 1 */
    n_2 = EXTRE(i,2);                /* nodo 2 */
    dx1 = TERMI(i,1);                /* division 1 */
    dx2 = TERMI(i,2);                /* division 2 */
     /* Calcula los nodos de cada segmento de linea
    por medio de una progresion geometrica */
    progresion(n_1, n_2, dx1, dx2);
    } /* fin de for i */
/*    getch();*/
    free(borde);            /* libera bloque de memoria */
    parametro = nnodosg; /* toma el parametro de nodos a refinar */
/*    vision(); */
} /* fin de funcion  cond_inic() */
#undef EXTRE
#undef TERMI

/* La siguiente rutina tiene por objeto convertir la region compuesta,
formada por uno o mas claros, en una simple, si ningun claro existe
saldra inmediatamente de esta. */
void preproceso()
 /*--------------*/
#define MAX(i) *(maxi+(i)-1)         /* Valor del arreglo nodos maximos */
#define MIN(i) *(mini+(i)-1)         /* Valor del arreglo nodos minimos */
#define MAY(i) *(mayo+(i)-1)         /* Valor del arreglo nodos mayores */
#define ACT(i) *(nodo_activo+(i))  /* Valor del arreglo nodos activos */
#define VIS(i) *(vision+(i)-1)       /* Nodo visible i */
{
    register int i, j, k;
    int nodo1, nodo_co, nodo_j;
    int *vision, clave, nodo_co2;
    int *divis;
    float dist_1, dist_2, dist_co, dist_j;
    int nodo_sg, claro;
    float angulo_mx=0, angulo_par;
```

```
void localiza();

if(nagujeros)    /* si hay por lo menos un claro... */
{
maxi = DEFM3(nagujeros);  /* Localiza espacio para nodos maximos */
mini = DEFM3(nagujeros);  /* Localiza espacio para nodos minimos */
mayo = DEFM3(nagujeros);  /* Localiza espacio para ordenar a maximos */
if(!maxi || !mini || !mayo)  ERROR; /* si no hay espacio entonces error */
localiza(); /* busca a los nodos maximo y minimo de cada claro */
            /* y los ordena a los maximos de mayor a menor */
/* Construye los conectores superiores parciales */
for(i=1; i<=nagujeros; i++)
{
    nodo1 = MAY(i);    /* nodo maximo del agujero */
    nod_visibles = 0;
    for(j=0; j<=nagujeros; j++)
    vision  = visibles(nodo1, ACT(j)); /* Nodos visibles */
    /* Toma el nodo con la menor distancia */
    angulo_mx = 0;   /* inicializar variables */
    dist_co   = 0;
    nodo_co2  = VIS(1);
    nodo_co   = VIS(1);
    for (j=1; j<=nod_visibles; j++)
    {
    dist_j = dist(nodo1, VIS(j));
    if( !dist_co ) dist_co = dist_j;
    if( dist_j < dist_co)
    {
        dist_co = dist_j;
        nodo_co = VIS(j);
    }
    angulo_par = ang_mx(nodo1, VIS(j));
    if(angulo_mx == 0) angulo_mx = angulo_par;
    if(angulo_mx <= angulo_par)
    {
        angulo_mx = angulo_par;
        nodo_co2 = VIS(j);
    }
    } /* fin de for j */
    clave = determinar(nodo_co); /* determina a que region pertenece */
    if(clave == 3 || !clave) /* no existe o es erroneo tal nodo */
    {
    nodo_co = nodo_co2;   /* cambiar a la segunda opcion */
    dist_co = dist(nodo1, nodo_co);
    clave   = determinar(nodo_co); /* region */
    if(clave == 3 || !clave)
    {
        printf("\n error en cadena*** \n");
        getch();
        closegraph();
        exit();
    }
    }
    if(clave == 1)  /* si pertenece a la frontera... */
    {
    n_cadenas++; /* incremento en el numero de cadenas */
    CADNA(n_cadenas, 1) = i; /* conector parcial */
    } /* fin de if */
    else if(clave == 2) /* pertenece a una cadena ya creada */
    {
```

```
        for(j=1; j<=n_cadenas; j++)   /* busca la cadena */
            for(k=1; k<=NUCOP(j); k++)
            if(global == COPAR(k,5))
                global = j;
        for(j=1; j<=nagujeros; j++)
            if(nodo_co == MIN(j))
            {  /* global es el numero de cadena al que pertenece */
            NUCOP(global)++; /* incremento en el numero de conectores */
            (cadena+global-1)->co_ptr = DEF33(
                (cadena+global-1)->co_ptr, NUCOP(global));
            CADNA(global, NUCOP(global)) = j; /* conec. parc. */
            break;
            }
        }

        for(j=1; j<=nagujeros; j++)   /* busca el numero del ajugero */
        if(nodo1 == MAX(j)) claro = j;

        COPAR(i,1) = nodo_co;       /* almacena los nodos del */
        COPAR(i,2) = *PN(nodo_co); /* conector parcial      */
        COPAR(i,3) = nodo1;
        COPAR(i,4) = *(PN(nodo1)-1);
        COPAR(i,5) = claro;
    } /* fin de for i */
    } /* fin de if */
} /* fin de la funcion preproceso */

void progresion(int nd_1, int nd_2, float d1, float d2)
   /*-------------*/
{
    int alma, anterior = 0;
    float fact, sf;
    float x1, y1, x2, y2, dx, dy, xb, yb;
    float coe[100], longi;
    int ndi[100];
    register int i;
/*    float x, y;  */

    nodos_pro = 0;              /* variable global inicializada */
    n_pri = nnodosg+1;          /* primer nodo de la progresion */
    x1   = COORD(nd_1, 1);
    y1   = COORD(nd_1, 2);
    x2   = COORD(nd_2, 1);
    y2   = COORD(nd_2, 2);
    dx   = x2 - x1;                 /* longitud en x del borde */
    dy   = y2 - y1;                 /* longitud en y del borde */
    longi = hypot(dx, dy);          /* longitud total */

        /* revizando condiciones */
    if(d1 <= 0 || d2 <= 0 || d1+d2 > longi)
    {
    nodos_pro = 0;    /* almacenar los nodos de los extremos */
    fact      = 1;
    }
    else
    {
    if(d1 == d2) d1 = d1 - d1/100;
    fact = (d1 -longi)/(d2 - longi);  /* proporcion de la progresion */
    if(fact == 1) fact = 1.00001;     /* evitando division entre cero */
    nodos_pro  = floor(log(d2/d1) / log(fact) +.5);        /* numero */
```

```
    if(nodos_pro > 100)
    {
        printf("\nel numero de nodos por lado fue excedido\n");
        printf(" aumentar su numero en el codigo fuente\n");
        printf(" progresion() max 100\n");
        getch();
        closegraph();
        exit(1);
    }
    }
    /* Calculo de coeficientes */
    for(i=0; i<= nodos_pro+1; i++)
    {
    if(i == 0)
    {
        coe[i] = 0.0;
        sf     = 1.0;
    }
    else
    {
        coe[i] = coe[i-1] + sf;
        sf     = sf * fact;
    }
    }
    sf = coe[nodos_pro+1];
    /* Calcula coordenadas */
    for(i=0; i<= nodos_pro+1; i++)
    {
    xb = dx * (coe[i]/sf) + x1;
    yb = dy * (coe[i]/sf) + y1;
        /* almacena las coordenadas  */
    alma = busqueda(xb, yb);  /* hay un nodo con estas coordenadas? */
    if(i!) alma = nd_1;
    if(i == nodos_pro+1) alma = nd_2;
    if(alma < 0)                          /* se crea un nodo */
    {
        nnodosg++;                /* incremento a nodos globales */
        nodos = DEF11(nnodosg); /* Crea espacio para nuevo nodo */
        if(!nodos)  ERROR;        /* Si no hay memoria disponible error */
        MAA(nnodosg) = 0;         /* inicializa */
        MEA(nnodosg) = 0;
        coord_ptr = DEF22(nnodosg); /* Crea espacio para coordenadas */
        if(!coord_ptr) ERROR;       /* nuevas a almacenar */
        COORD(nnodosg,1) = xb;  /* Almacena a x */
        COORD(nnodosg,2) = yb;  /* Almacena a y */
        alma = nnodosg;       /* toma el numero asignado al nodo */
    }
    if(anterior)       /* si el nodo anterior es diferente de cero */
        adyacentes(anterior, alma); /* inicializa las listas adyacentes */
        anterior = alma; /* de lo contrario tomar el nodo presente como */
                         /*  anterior  */
    ndl[i] = alma;
    } /* fin de for i */
    n_seg = nnodosg;              /* ultimo nodo de la progresion */
    linea(nd_1, nd_2);           /* trazo de linea */
    return;
} /* fin de funcion progresion */

void adyacentes(int nodal1, int nodal2)
 /*------------*/
```

```
{
    int n1, n2;
    void incremento(int);

    if(!NNA(nodal1) ) inic(nodal1);/* inicializa sino lo ha sido*/
    if(!NNA(nodal2) ) inic(nodal2);
    NNA(nodal1)++;  /* incremento en el numero de nodos adyacentes*/
    NNA(nodal2)++;
    if(NNA(nodal1) > 2) incremento(nodal1);
    if(NNA(nodal2) > 2)
    {
    incremento(nodal2);
    PN(nodal2)++;
    }
    *PN(nodal1)     = nodal2;
    *(PN(nodal2)-1) = nodal1;
)/* fin de funcion adyacentes */


void alterar(int nd_1, int nd_2)
  /*---------*/
{
    int nu_a1, nu_a2;
    register int i;

    nu_a1 = NNA(nd_1); /* Numero de nodos adyacentes del nodo 1 */
    nu_a2 = NNA(nd_2); /* Numero de nodos adyacentes del nodo 2 */
    /* busca espacio en la memoria dinamica */
    if(nu_a1 == 0)
    PN(nd_1)=(nodos+nd_1-1)->nodosa = DEFM3(2); /* inicializar */
    else
    {
    /* diferencia de direccion */
    diferencia = (nodos+nd_1-1)->nodosa;
    (nodos+nd_1-1)->nodosa =
        DEF33((nodos+nd_1-1)->nodosa, nu_a1+1);  /* incrementar */
    PN(nd_1) = PN(nd_1)-diferencia+(nodos+nd_1-1)->nodosa;
    }
    if(nu_a2 == 0)
    PN(nd_2)=(nodos+nd_2-1)->nodosa = DEFM3(2);      /* inicializar */
    else
    {
    /* diferencia de direccion */
    diferencia = (nodos+nd_2-1)->nodosa;
    (nodos+nd_2-1)->nodosa =
        DEF33((nodos+nd_2-1)->nodosa, nu_a2+1);  /* incrementar */
    PN(nd_2) = PN(nd_2)-diferencia+(nodos+nd_2-1)->nodosa;
    }
    if(!(nodos+nd_1-1)->nodosa || !(nodos+nd_2-1)->nodosa) ERROR;
    /* Cambia de lugar a los nodos adyacentes */
    for(i=nu_a1; i>=1; i--)
    {
    NDNA(nd_1,i+1) = NDNA(nd_1, i); /* cambia de posicion al nodo */
    if(*PN(nd_1) == NDNA(nd_1, i)) /* adyacente hasta encontrar al nodo */
        break;                     /* que apunta el puntero */
    }
    for(i=nu_a2; i>=1; i--)
    {
    NDNA(nd_2, i+1) = NDNA(nd_2, i);
    if(*PN(nd_2) == NDNA(nd_2, i))
```

```
        break;
    }
    /* Inserta a nd_1 y nd_2 en los lugares en que apunta el puntero*/
    *PN(nd_1) = nd_2;
    *PN(nd_2) = nd_1;
    NNA(nd_1)++;    /* incremento en el numero de nodos adyacentes */
    NNA(nd_2)++;
    if(NNA(nd_2) > 1)
    PN(nd_2)++;    /* mover el puntero dos a la siguiente posicion */
} /* fin de funcion alterar */

float angulo(int n0, int n1, int n2)
  /*----------*/
{
    float x0, y0, x1, y1, x2, y2;
    float  ang_i;

    if(n0 == n1 || n0 == n2 || n2 == n1)
    return(0.0);
    x0 = COORD(n0, 1);
    y0 = COORD(n0, 2);
    x1 = COORD(n1, 1);
    y1 = COORD(n1, 2);
    x2 = COORD(n2, 1);
    y2 = COORD(n2, 2);
    /* eje de coordenadas local */
    x0 = x0 - x1;
    y0 = y0 - y1;
    x2 = x2 - x1;
    y2 = y2 - y1;
    /* checar division entre cero */
    if(x0 == 0) x0 = 0.000000001;
    if(x2 == 0) x2 = 0.000000001;
    /* Calcula los angulos con respecto al eje x local */
    ang1 = atan(y0/x0)*GRA + cuadrante(x0,y0); /* primer angulo */
    ang2 = atan(y2/x2)*GRA + cuadrante(x2,y2); /* segundo angulo */
  /*  comb = 0;*/
    /* Calcula el angulo entre los tres puntos */
    if(ang1 > ang2)
    ang_i = 360 - ang1 + ang2;
    else
    ang_i = ang2 - ang1;
    return(ang_i);
} /* fin de funcion angulo */

/* La siguiente rutina genera los nodos visibles para un nodo */
int *visibles(int nprincipal, int nactivo)
  /*----------*/
{
    register int i, k;
    static int vistos[200]; /* arreglo nodos visibles */
    int *no_ptr1;           /* puntero de nodos visibles */
    float angulos_e[200], *an_ptr2; /* angulos de nodos visibles */
    int nodo_i, num1;
    float ang_ant = 0.0;  /* angulo anterior al nodo i */
    int nanterior, nposterior; /* nodo anterior y posterior */
    float angular;          /* angulo del nodo pivote  */
    float ang_p, ang_med, ang_may;
    float ang_i;    /* angulo en proceso */
    float dist_i, dist_a; /* distancia del nodo actual y anterior */
```

```
int nodo_ant=0; /* nodo anterior al nodo i */
float gama_nv; /* nuevo angulo */
float ang_men;
int acep, nodosvp = 0, parcial;


an_ptr2   = angulos_a;  /* apuntador de angulos */
no_ptr1 = vistos + nod_visibles; /* apuntador de nodos visibles */
nodo_i = nactivo;  /* nodo que activa la region */
nanterior = transferencia(nprincipal, -1); /* nodo anterior */
nposterior = transferencia(nprincipal,  1); /* nodo posterior */
angular =angulo(nanterior, nprincipal, nposterior); /* angulo de vision */
/* Calcula los rangos minimo y maximo del nodo pivote
   en la base del valor de GAMA */
if(angular > 2*GAMA)
{
num1    = ceil(angular/GAMA);
gama_nv = angular/num1; /* recalcula gama con num1 redondeando a este */
}
else
gama_nv = angular/2;
/* ang1 y ang2 son valores globales del angulo pivote */
ang_med = ang1 + angular/2;
/* determinando limites de angulo de vision */
ang_men = ang_med - gama_nv;  /* limite minimo */
ang_may = ang_med + gama_nv;  /* limite maximo */
ang_ant = ang_may;
 /* Calcula el numero de nodos visibles para el nodo pivote */
do
{
ang_i = angulo(nanterior, nprincipal, nodo_i); /* angulo del nodo */
ang_i += ang1;  /* toma el segundo limite del angulo interior */
if(ang_i > ang_men+1) /* comprueba que el angulo este dentro del rango */
    if(ang_i < ang_may-1)
    {
    if(ang_i - ang_ant < 0)
       acep =1;       /* incremento positivo del angulo */
    else              /* de lo contrario un nodo no es visible */
    {
       dist_i = dist(nprincipal, nodo_i);  /* toma al que */
       dist_a = dist(nprincipal, nodo_ant); /* tenga menor */
       if(dist_i < dist_a)                  /* distancia */
       { /* si la distancia al nodo es menor que la distancia
        del nodo anterior buscara los angulos menores a
        este nodo  */
       parcial = nod_visibles;
       for(k=1; k<nodosvp; k++)
          if(*(an_ptr2-k)-ang_i <= 0)
          {
          no_ptr1--;  /* restarlos de la lista */
          an_ptr2--;  /* ya que no son visibles */
          nod_visibles--;
          nodosvp--;
          }
       nodosvp -= parcial-nod_visibles;
       acep = 1;  /* se toma como visible */
             } /* fin de if */
       else
       acep = 0;   /* no se acepta como visible */
    }  /* fin de else */
```

```
        if(acep) /* si se cumple almacena al nodo */
        {
            *no_ptr1 = nodo_i; /* nodo almacenado */
            *an_ptr2 = ang_i;  /* angulo nodo almacenado */
            no_ptr1++;   /* incremento de visibles */
            an_ptr2++;   /* incrementos de angulos */
            nod_visibles++; /* numero de vertices visibles */
            if(nod_visibles > 200)
            {
            closegraph();
            printf("\nNo hay espacio suficiente para\n");
            printf("almacenar los nodos vistos.\n");
            printf("Ampliar los arreglos static vistos[]\n");
            printf("y angulos_a[].\n");
            getch();
            exit();
            }
            nodosvp++;      /*  numero de vertices visibles parcial */
        }
        } /* fin de if */
    ang_ant = ang_i;
    nodo_ant = nodo_i;
    nodo_i = transferencia(nodo_i, 1);
    if(nodo_i == nprincipal)
        nodo_i = transferencia(nodo_i, 1);
    } while(nodo_i != nactivo);
    return(vistos);
} /* fin de la funcion visibles */

/* La siguiente rutina tiene el objetivo de introducir el control
   del programa al proceso de generacion de elementos triangulares */
void proceso()
  /*-----------*/
{
    int ncp = 0, cp = 0 ;
    register int i;

    if(nagujeros)
    descomponer(1, ncp);  /* si hay agujeros */
    else
    dividir(1, cp);       /* si no hay agujeros */
    for(i=parametro; i<=nnodosag; i++)
    refinado(i, transferencia(i, 1));
    limpiar();
    cla_ve = getch();  /* el proceso fué completado */
    if(cla_ve == 'i') LJ_Graphic(0);   /* imprimir(); */
/*    imprimir();  */
    getch();
    closegraph();
}

void descomponer(int nodo_activo1, int ncp)
  /*-------------*/
{
    register int i, j;
    int nodo_my_i;      /* nodo mayor i */
    int lugar = 0;      /* lugar ocupado por el agujero */
    int nagup = 0;      /* numero de agujeros parciales */
    int *agu_ap;        /* apuntador de lista de agujeros de la region */
    float xmin=0, xmax=0, ymin=0, ymax=0; /* limites de la subregion */
```

```
    int nod_p = 0, nodo_co;
    int nodal, *vistos, *vision;
    float x_np, y_np;
    int numes;
    float dist1, dist_co;
    float dist_1=0, dist_2;
    int qp, num_agu;
    int *division;
    int nopo1, nopo2, conector_p;

    dist_co = 0;
    nod_p = nodo_activo1;
/* Comprobara si en la region hay un agujero */
    xmin = COORD(nod_p, 1);
    ymin = COORD(nod_p, 2);
/* Busca los limites de la region */
    do
    {
    x_np = COORD(nod_p, 1);
    y_np = COORD(nod_p, 2);
        if(xmin > x_np) xmin = x_np;
    if(xmax < x_np) xmax = x_np;
    if(ymin > y_np) ymin = y_np;
    if(ymax < y_np) ymax = y_np;
    nod_p = transferencia(nod_p, 1); /*  siguiente nodo activo */
    }while(nodo_activo1 != nod_p);
    /* Compara si los nodos del agujero se ecuentran fuera de la region */
    for(i=1; i<=nagujeros; i++)
    {
    nod_p = transferencia(MAX(i), 1);
    nodal = MAX(i);
    do
        {
        x_np = COORD(nod_p,1);
        y_np = COORD(nod_p,2);
        if(x_np <= xmin) break; /*  si se cumple pasara al      */
        if(x_np >= xmax) break; /*  siguiente agujero ya que    */
        if(y_np <= ymin) break; /*  significa que este          */
        if(y_np >= ymax) break; /*  fuera de la region          */
            /* cambia al siguiente nodo */
        nod_p = transferencia(nod_p, 1);
    }while(MAX(i) != nod_p);
    if(nod_p == MAX(i)) /* si se cumple existe por lo menos un ajugero */
    {
        nagup++;                /* encontro un agujero en la region */
        if(nagup < 2)
        agu_ap = DEF33(agu_ap, nagup);
        else
        agu_ap = DEF33(agu_ap, nagup);
        if(!agu_ap) ERROR;      /* no hay memoria disponible */
        *(agu_ap+nagup-1) = i;   /* almacena su etiqueta */
    }
    } /* fin de for i */
    if(nagup) /* si el numero de agujeros de la subregion no es cero...*/
    {
    lugar = 0;
    for(i=1; i<=nagup; i++)
    {
     /* toma el nodo maximo del agujero */
        nodo_my_i = MAX(*(agu_ap+i-1));
```

```
    /* busca el menor de los maximos */
       for(j=1; j<=nagujeros; j++)
       if(nodo_my_i == MAY(j))
           if(j > lugar)
           {
           lugar = j;  /* lugar del orden mayor a menor */
           nodal = MIN(*(agu_ap+i-1)); /* nodo menor */
           num_agu = *(agu_ap+i-1); /* numero de agujero */
           break; /* siguiente agujero parcial */
           }
    }
    /* nodos visibles para este nodo */
    nod_visibles = 0;
    vistos =  visibles(nodal, 1);
    vision = vistos;
    conec_parc = realloc(conec_parc,(nagujeros+ncp+1)*sizeof(struct cuatro));
    if(!conec_parc) ERROR; /* no hay memoria disponible */
    ncp++;                 /* aumenta el numero de conectores */
    /* toma el nodo con menor distancia */
    nodo_co = *vistos;
    for(i=0; i<nod_visibles; i++)
    {
        disti = dist(nodal, *(vistos+i));
        if(dist_co == 0)  dist_co = disti;
        if(disti < dist_co)
        {
        nodo_co = *(vision+i);
        dist_co = disti;
        }
    }/* fin de for i */
    /* busca la cadena en donde almacenar el conector parcial */
    for(i=1; i<=n_cadenas; i++)
        for(j=1; j<=MUCOP(i); j++)
        if(COPAR(j,5) == num_agu)
            lugar = i;
    MUCOP(lugar)++; /* incremento en el numero de conectores */
    (cadena+lugar-1)->co_ptr = DEF33(
                (cadena+lugar-1)->co_ptr, MUCOP(lugar));
    CADNA(lugar, MUCOP(lugar)) = nagujeros+ncp; /* conec. parc. */
    qp = nagujeros+ncp;
    COPAR(qp,1) = nodal;
    COPAR(qp,2) = *PM(nodal);
    COPAR(qp,3) = nodo_co;
    COPAR(qp,4) = *(PM(nodo_co)-1);
    COPAR(qp,5) = 0 ; /* el nodo pertenece a frontera del dominio */
    /* divide la distancia del conector deacuerdo con las distancias
       promedio de sus extremos */
/*  progresion(nodal, nodo_co, dist_1, dist_2);*/
    /* activa la cadena */
    for(i=1; i<=MUCOP(lugar);i++)
    {
        conector_p = CADNA(lugar, i);  /* conector parcial i */
        nopo1  = COPAR(conector_p, 1);
        nopo2  = COPAR(conector_p, 3);
        dist_1 = p_dis(nopo1);
        dist_2 = p_dis(nopo2);
        progresion(nopo1, nopo2, dist_1, dist_2);
    }
    descomponer(nodal,0); /* revisa la region resultante */
    /* calculo el conector parcial superior */
```

```
    } /* fin de if */
    else
    dividir(nodal, 0);  /*uno*/
    if(incp)  return;
    /* Reordenar la cadena ncp en direccion reversiva */
    for(i=1; i<=NUCOP(lugar);i++)
    {
    conector_p = CADNA(lugar, i);  /* conector parcial i */
    reordenar1( (conec_parc+conector_p-1)->s );
    }
    ncp--;
    descomponer(nodal, 0);
} /* fin de funcion descomponer */

void dividir (int nodo, int nc)
  /*----------*/
{
    float angular, ang_ele=0, angulo_max = 0;
    int nodo_p, n_mx=0;
    int *nodos_visi;
    int nodo_p0, nodo_p1;
    register int i;
    float dist_1, dist_2;
    int conec[4];
    int  nodo_ele;
    int trans;
    /* revisa si la region es convexa */
    nodo_p = nodo;
    do
    {
    nodo_p0 = transferencia(nodo_p, -1);
    nodo_p1 = transferencia(nodo_p,  1);
    angular = angulo(nodo_p0, nodo_p, nodo_p1);
    if(angular > angulo_max)
    {
        angulo_max = angular;
        n_mx    = nodo_p;
    }
    nodo_p = nodo_p1;
    if(nnodosg < nodo_p)
    {
        printf("\n El numero de nodo trasferido\n");
        printf(" es mayor al generado ciclo infinito\n");
        getch();
        closegraph();
        exit(1);
    }
    } while(nodo_p != nodo);
    if(angulo_max > 185) /* Si se cumple, la region es convexa */
    {
    /* Crea una nueva subregion activa */
    /* nodos visibles */
    nod_visibles = 0;       /* inicializa la variable */
    trans = transferencia(n_mx, 1);
    nodos_visi = visibles(n_mx, trans); /* nodos visibles */
    if (!nod_visibles)   /* si no hay nodos visibles...*/
    {
        printf("\n no hay nodos visibles ");
        getch();
        return;
```

```
    }

    for(i=0; i<nod_visibles; i++)
    {
        nodo_p = *(nodos_visi+i);
        angulo_max = ang_mx(n_mx, nodo_p); /* busca el angulo */
        if(angulo_max > ang_ele)      /*angulo maximo de los minimos */
        {
        ang_ele = angulo_max;
        nodo_ele   = nodo_p;
        } /* fin de if */
    } /* fin de for */
    dist_1 = p_dis(n_mx);
    dist_2 = p_dis(nodo_ele);

    /* Crea el conector  */
    n_co++;
    nc++;
    conec[0]  = n_mx;                /* extremo uno */
    conec[1]  = *PN(n_mx);           /* adyacente al extremo uno */
    conec[2]  = nodo_ele;            /* extremo dos */
    conec[3]  = *(PN(nodo_ele)-1); /* adyacentes al extremo dos */
        /* divide la distancia del conector de acuerdo con las
           distancias promedio */

    progresion(n_mx, nodo_ele, dist_1, dist_2);

    CONEC(n_co, 1) = n_mx;      /* nodo inicial del conector */
    CONEC(n_co, 2) = n_pri;     /* nodo primero de la progresión */
    CONEC(n_co, 3) = nodo_ele; /* nodo final del conector */
    CONEC(n_co, 4) = n_seg;     /* nodo último de la progresión */

    dividir(n_mx, 0); /* probar que la region resultante
                sea convexa */
    } /* fin de if */
    else
    emllar(nodo); /* si la region es convexa triangular */
    if(inc) return;
    /* Definir una nueva subregion activa por el reordenado
       del conector n_c */
    reordenar1(&conec[0]); /* reordenar el conector */
    n_co--;
    dividir(conec[0], 0);
} /* fin de funcion dividir */

void mallar(int n_activo)
  /*--------*/
{
    int n_anterior, n_posterior;
    float an_interior;
    int bis, falso_n;

    falso_n = transferencia(n_activo, -1);
    for(;;)
    {
    /* si la region es un triangulo termina el proceso */
    n_anterior  = transferencia(n_activo, -1); /* toma el angulo anterior */
    n_posterior = transferencia(n_activo,  1); /* toma el angulo posterior */
    /* calcula el angulo interior */
    if(transferencia(n_activo, 3) == n_activo)
```

```
    {       /* inicializa y almacena */
        almacen(n_anterior, n_activo, n_posterior);
/*      checar(n_anterior);  revisa el numero de elementos adyacentes */
    /*    checar(n_activo);
        checar(n_posterior);*/
        break;
    }
    an_interior  = angulo(n_anterior, n_activo, n_posterior);
    if(an_interior <= ANG_MN) /* si se cumple "cierra" el angulo */
    {
        cerrar(n_anterior, n_activo, n_posterior);
        clavel = 1;
    }
    else if(an_interior <= ANG_MY) /* crea un nuevo nodo */
    {
        bisectar(n_anterior, n_activo, n_posterior);
        if(!clavel)
        if(n_activo == falso_n)
        {
            dividir2(falso_n, 0);
            return;
        }
    }
    else  /* no encontro un angulo adecuado */
    {
        clavel = 0;
        if(falso_n == n_activo)
        {
        /* se recorrieron todos los nodos sin
             encontrar un angulo adecuado */
        dividir2(falso_n, 0);  /* numero de c */
        return;
        }
    }
    if(clavel) falso_n = transferencia(n_activo, -1);
    n_activo = transferencia(n_activo, 1);
    } /* fin de for infinito */
/*    imprimir(); */
} /* fin de funcion emllar() */

void cerrar(int n0, int n1, int n2)
 /*---------*/
{
    almacen(n0, n1, n2);  /* inicializacion y almacenamiento */
    alterar(n0, n2); /*altera ambas listas nodo-nodo adyacente */
    linea(n0, n2); /* trazo de linea */
    checar(n0);/* revisa numero de nodos adyacentes */
 /* checar(n1); */
    checar(n2);
} /* fin de funcion cerrar */

void bisectar(int nd0, int nd1, int nd2)
  /*-----------*/
{
    int *vistos, aceptar=0;
    float x0, y0, x1, y1, x2, y2;
    float xm, ym;
    register int i;
    int ndp, ndp0, ndp1;
    float distp, avrg;
```

```
    float *ptr_rt;

    x0 = COORD(nd0, 1);
    y0 = COORD(nd0, 2);
    x1 = COORD(nd1, 1);
    y1 = COORD(nd1, 2);
    x2 = COORD(nd2, 1);
    y2 = COORD(nd2, 2);
/*  Rotar el primer lado sesenta grados para obtener las coordenadas
    del triangulo equilatero del lado 1-0  en contra de las manecillas
    del reloj. */
    ptr_rt = rotar(x1, y1, x0, y0, 1);
    x0 = *ptr_rt;
    y0 = *(ptr_rt+1);
    /* rotar el segundo lado  menos sesenta grados para obtener
       las coordenadas del triangulo equilatero del lado 1-2 en el
       sentido de las manecillas del reloj */
    ptr_rt = rotar(x1, y1, x2, y2, -1);
    x2 = *ptr_rt;
    y2 = *(ptr_rt+1);
    /* calcula la media entre las dos coordenadas */
    xm = (x2-x0)/2 + x0;
    ym = (y2-y0)/2 + y0;
    /* calcula los nodos visibles para nd1 para
       aceptar o rechazar el punto medio  */
    nod_visibles = 0;
    vistos = visibles(nd1, transferencia(nd1,1)); /* nodos visibles para nd1 */
    aceptar = 0;
    for(i=0; i<nod_visibles; i++)
    {
    ndp   = *(vistos+i);  /*toma el nodo visible */
    x0    = COORD(ndp, 1);
    y0    = COORD(ndp, 2);
    distp = sqrt( pow((x0-xm),2) + pow((y0-ym),2) );
    ndp0  = transferencia(ndp, -1);
    ndp1  = transferencia(ndp,  1);
    avrg  = 0.433 * (dist(ndp, ndp1)+dist(ndp, ndp0));
    if(avrg < distp && nod_visibles > 1) aceptar++;
    else
       break; /* salir si no se cumple para todos */
    } /* fin de for i */
    if(aceptar == nod_visibles)  /* si es igual aceptar */
    {
    nnodosg++;
    nodos = DEF11(nnodosg); /* Crea espacio para nuevo nodo */
    if(!nodos)  ERROR;      /* Si no hay memoria disponible error */
    NMA(nnodosg) = 0;       /* inicializa */
    NEA(nnodosg) = 0;
    coord_ptr = DEF22(nnodosg); /* Crea espacio para coordenadas */
    if(!coord_ptr) ERROR; /* nuevas a almacenar */
    COORD(nnodosg,1) = xm;
    COORD(nnodosg,2) = ym;
    alterar(nd1, nnodosg); /* nd1 queda fuera de actividad */
    alterar(nnodosg, nd2);
    alterar(nd0, nnodosg);
    linea(nnodosg, nd0);  /* primera linea */
    linea(nnodosg, nd1);  /* segunda linea */
    linea(nnodosg, nd2);  /* tercera linea */
    almacen(nnodosg, nd0, nd1); /* inicializa y almacena */
    almacen(nnodosg, nd1, nd2);
```

```
    checar(nnodosg);  /* revisa el numero de elementos adyacentes */
    checar(nd0);
    checar(nd2);
    clavei = 1;  /* se realizo con exito la biseccion */
    }
    else
    clavei = 0;   /* no se realizo el proceso de biseccion */
}/* fin de funcion bisectar */

/* La siguiente rutina tiene como objetivo localizar los nodos
   maximos y minimos que se encuentran en cada claro o agujero
   del dominio  */

void localiza()
  /*----------*/
{
    int nodo_n;
    register int i;
    int casilla, parcial, temporal;
    for(i=1; i<= nagujeros; i++)
    {
/*      Busca al nodo cuyas coordenadas sean mayores
        y menores en cada agujero tomando en cuenta:
        (x1,y1) < (x2,y2) si y1 < y2   o   y1 = y2   y   x1 < x2  */
    MIN(i) = ACT(i);   /* inicializa al conjunto minimo */
    MAX(i) = ACT(i);   /* inicializa alconjunto mayor-menor */
    nodo_n = ACT(i);   /* toma al primer nodo activo del agujero */
        do
    {       /* Busca al maximo y minimo */
        if(COORD(nodo_n, 2) > COORD(MAX(i), 2))
        {
        MAX(i) = nodo_n;
        MAY(i) = nodo_n;
        continue;
         }
         else if( COORD(nodo_n, 2) == COORD(MAX(i), 2) )
        if(COORD(nodo_n, 1) > COORD(MAX(i), 1))
        {
            MAX(i) = nodo_n;
            MAY(i) = nodo_n;
            continue;
        }
            /* Busca el nodo minimo */
        if(COORD(nodo_n, 2) < COORD(MIN(i),2))
        MIN(i) = nodo_n;
        else if(COORD(nodo_n, 2) == COORD(MIN(i), 2))
        {
        if(COORD(nodo_n, 1) < COORD(MIN(i), 1))
        MIN(i) = nodo_n;
        }
            nodo_n = transferencia(nodo_n, 1);
        } while(ACT(i) != nodo_n);
    } /* fin de for i */
    /* Ordena a los nodos maximos de mayor a menor */
    casilla = 1;
    do
    {
    parcial = MAY(casilla); /* toma al nodo de la primera casilla */
    for(i=casilla; i<=nagujeros; i++)
        if(COORD(parcial, 2) < COORD(MAY(i), 2))  /* compara ordenadas */
```

```
        {
        temporal = parcial;  /* si la nueva ordenada es mayor cambiar */
        parcial  = MAY(i);  /* valores y dejar en esta casilla */
        MAY(i)   = temporal; /* anterior ordenada */
        MAY(casilla) = parcial; /* dejar en la ultima casilla a la ordenada */
        }
    casilla++;              /* mayor tomar la siguiente casilla */
    } while(casilla <= nagujeros); /* recorridas todas las casillas salir */
} /* fin de funcion localizar() */


void cambio(int nd1, int nd2)
{
    register int i;

    if(*PN(nd1) != nd2)   /* observa si hay que hacer cambios */
    {
    for(i=1; i<=NNA(nd1); i++)   /* busca el lugar que ocupa */
    {
        if(nd2 == NDNA(nd1,i))
        {
        NDNA(nd1, i) = *PN(nd1); /* cambio de lugar */
        *PN(nd1) = nd2;
        break; /* salir del ciclo */
        }
    }
    }
    if(*(PN(nd2)-1) != nd1)
    {
    for(i=1; i<=NNA(nd2); i++)
    {
        if(nd1 == NDNA(nd2,i))
        {
        NDNA(nd2,i)  = *(PN(nd2)-1);
        *(PN(nd2)-1) = nd1;
        break;
        }
    }
    }
} /* fin de funcion cambio() */

/* La siguiente rutina determina a que region pertenece el nodo nd */
int determinar(int nd)
 /*------------*/
{
    int nodo = 1, nodal;
    register int i, j;

    nodal = transferencia(nodo, 1);
    nodo  = nodal;
    do
    {
    if(nodo == nd)  return(1);  /* pertenece a la frontera */
    nodo = transferencia(nodo, 1);
    }while(nodo != nodal);
    for(i=1; i<=nagujeros; i++)
    {
    nodal = transferencia(*(nodo_activo+i), 1);
    nodo  = nodal;
    do
```

```
    {
        if(nodo == nd)
        {
        for(j=1; j<=nagujeros; j++)
        {
            if(nodo ==MIN[j])
            {
            global = j; /* almacena la etiqueta */
            return(2); /* es un nodo minimo de un agujero */
            }
        }
        return(3); /* es un nodo de cualquier agujero */
        }
        nodo = transferencia(nodo, 1);
    } while(nodo != nodo1);
    }/* fin de for i */
    return(0); /* no pertenece a ninguna region error */
} /* fin de la funcion determinar() */

void inic_graph(float dx1, float dy1)
  /*-------------*/
{
    int g_driver, g_mode, g_error;
    float dx2, dy2;

    detectgraph( &g_driver, &g_mode);       /* llama a los dispositivos */
    initgraph( &g_driver, &g_mode, "c:\\tc");       /* inicializa */
    g_error = graphresult();
    if(g_error < 0)
    {
    printf("\n error grafico\n");
    getch();
    exit(1);
    }
    cleardevice();  /* limpia la pantalla */
    /* calcula los parametros a usar en la pantalla de
     acuerdo con la figura a desplegar  */
    dy2 = dy1 + dy1 * 0.1; /* aumenta el diez por ciento */
    dx2 = PX * dy2/PY;      /* pixel cuadrado */
    if(dx2 < dx1)  /* si se cumple el eje x no cabra en la pantalla */
    {
    dx2 = dx1 + dx1 * 0.1;    /* cambiar parametros en funcion de x */
    dy2 = PY * dx2 / PX;
    }
    esc = PY/dy2;               /* escala */
    ddx = (dx2 - dx1)/2;
    ddy = (dy2 - dy1)/2;
} /* fin de inic_graph () */

void dividir2 (int nodo, int nc)
  /*----------*/
{
    float angular, ang_ele=0, angulo_max = 0;
    int nodo_p, n_ax=0;
    int *nodos_visi;
    int nodo_p0, nodo_p1;
    register int i;
    float dist_1, dist_2;
    int conec[4];
    float distan;
```

```
int nodo_ele;

/* toma el angulo mayor */
nodo_p = nodo;
do
{
nodo_p0 = transferencia(nodo_p, -1);
nodo_p1 = transferencia(nodo_p,  1);
angular = angulo(nodo_p0, nodo_p, nodo_p1);
if(angular > angulo_max)
{
    angulo_max = angular;
    n_mx   = nodo_p;
}
    nodo_p = nodo_p1;
} while(nodo_p != nodo);
/* Crea una nueva subregion activa */
/* nodos visibles */
nod_visibles = 0;       /* inicializa la variable */
nodos_visi = visibles(n_mx, transferencia(n_mx,1)); /* nodos visibles */
if (!nod_visibles)   /* si no hay nodos visibles...*/
{
printf("\n no hay nodos visibles en dividir2 ");
putpixel(X(COORD(n_mx,1)), Y(COORD(n_mx,2)), 0 );
getch();
closegraph();
exit(1);
return;
}
for(i=0; i<nod_visibles; i++)
{
nodo_p = *(nodos_visi+i);
angulo_max = ang_mx(n_mx, nodo_p); /* busca el angulo */
if(angulo_max > ang_ele)           /*  maximo de los minimos */
{
    nodo_ele = *(nodos_visi+i);
    ang_ele = angulo_max; /* cambia angulo maximo de los minimos */
} /* fin de if */
} /* fin de for */
dist_1 = p_dis(n_mx);  /* busca distancia promedio en los */
dist_2 = p_dis(nodo_ele); /* nodos de los extremos  */
    /* Crea el conector estático */
conec[0]  = n_mx;                  /*extremo uno*/
conec[1]  = *PW(n_mx);             /* adyacente al extremo uno */
conec[2]  = nodo_ele;              /* extremo dos */
conec[3]  = *(PW(nodo_ele)-1);     /* adyacentes al extremo dos */
    /* divide la distancia del conector de acuerdo con las
       distancias promedio */
progresion(n_mx, nodo_ele, dist_1, dist_2);  /* crea nuevos nodos */
linea(n_mx, nodo_ele);                       /*  traza linea  */
n_co++;
if(n_co == 15)
{
printf(" \n no hay memoria para conectores \n");
getch();
closegraph();
exit();
}
CONEC(n_co-1)  = n_mx;     /* nodo inicial del conector */
CONEC(n_co-2)  = n_pri;    /* nodo primero de la progresion */
```

```
    CONEC(n_co, 3) = nodo_ele; /* nodo final del conector */
    CONEC(n_co, 4) = n_seg;    /* nodo último de la progresión */
    mallar(n_mx);
    reordenar1(&conec[0]); /* reordenar el conector */
    n_co--;                /* al reordenar  disminuye */
    mallar(conec[0]);
} /* fin de funcion dividir2 */

void inic_elem(int nodoe)
{
    if(!NEA(nodoe))
    {
    PE(nodoe) = (nodos+nodoe-1)->elementos = DEFM3(2);
    NEA(nodoe)++;
    }
    else
    {
    /* diferencia de direccion */
    NEA(nodoe)++;
    PE(nodoe)=(nodos+nodoe-1)->elementos =
        DEF33((nodos+nodoe-1)->elementos, NEA(nodoe));
    }
    if(!PE(nodoe)) ERROR;
} /* fin de inic_elem() */

void incremento(int nodo_f)
{
    register int i;

    diferencia = (nodos+nodo_f-1)->nodosa;
    (nodos+nodo_f-1)->nodosa =
    DEF33((nodos+nodo_f-1)->nodosa, NNA(nodo_f));
    PN(nodo_f) = PN(nodo_f)-diferencia+(nodos+nodo_f-1)->nodosa;
      /* Cambia de lugar a los nodos adyacentes */
    for(i=NNA(nodo_f)-1; i>=1; i--)
    {
    NDNA(nodo_f,i+1) = NDNA(nodo_f, i); /* cambia de posicion al nodo */
    if(*PN(nodo_f) == NDNA(nodo_f, i)) /* adyacente hasta encontrar al nodo */
        break;                         /* que apunta el puntero */
    }
}

void checar(int nd_0)
{
    if(NEA(nd_0) > 2) refinado(nd_0, transferencia(nd_0, 1));
}

main()
{
    leer();
    cond_inic();
    preproceso();
    proceso();
    resumen();
}
```

```
/* Programa generador de elementos cuadrangulares
   para el método del elemento finito. (nuevo3.c)   */


#include <stdio.h>     /* archivos estandar del compilador */
#include <graphics.h>
#include <alloc.h>
#include <stdlib.h>
#include <conio.h>
#include <dos.h>
#define CAMP         07  /* campana */
#define NUM_NODOS    72   /* Numero de nodos iniciales */
#define NUM_CARAS    52   /* Numero de caras iniciales */
#define ESC_X        17   /* Escala en eje x */
#define ESC_Y        17  /* Escala en eje y */
#define COOR(n,i,j) ((proc[(n)].coordenadas.nodo+(i)-1)->xy[(j)])
#define ELEM(n,i,j) ((proc[(n)].nodos.elemento+(i)-1)->vert[(j)-1])
#define ETIQ(n,i,j) ((proc[(n)].etiquetas.cara+(i)-1)->vert[(j)-1])
#define NOD(i)      (*(node+(i)-1))
#define NUM(i)      (*(numa_1+(i)-1))
#define BDE(i,j,k)  ((rostro+(i)-1)->borde[(j)-1][(k)])   /* rostro-borde-etiqueta */
#define PROV(n,i,j) (*(prov+(n-1)*10+(i)*2+(j)))
#define NLA(i)      (*(numle+(i)-1))     /* numero de lados adyacentes */
#define NNA(i,j)    ((nodo_j+(i)-1)->adyacente[(j)-1]) /* numero de nodos adyacentes */

          /* definicion de estructuras */
struct dos
{
    float xy[2];
};

struct cuatro
{
    int vert[4];
};

struct arreglo_1
{
    struct cuatro *cara;
};

struct arreglo_2
{
    struct dos *nodo;
};

struct arreglo_3
{
    struct cuatro *elemento;
};

struct cercano
{
    int adyacente[5];
};

struct statous
{
    struct arreglo_1 etiquetas;
    struct arreglo_2 coordenadas;
```

```
    struct arreglo_3 nodos;
} proc[2];

struct simple
{
    int nivel;
};

struct adya
{
    int borde[4][2];
};

                /* Variables globales */

    int *prov;    /* puntero de almacenamiento provisional */
    int *nume_1;  /* puntero de numero de caras NUM() */
    int sen, apu;
    float *node;          /* puntero de nivel maximo de division */
    float coord[10][2]; /* coordenadas del elemento en proceso */
    int etiqueta[10];   /* etiquetas del elemento en proceso */
    int numero_rostros; /* elementos generados en el proceso */
    int numero_nodos1 = NUM_NODOS; /* nodos iniciales */
    int numero_nodos2 = 1;  /* contador de elementos en proceso */
    int elemento[5][5];  /* elementos del proceso parcial */
    int numero_caras = NUM_CARAS; /* numero de elementos inicial */
    int *numla;  /* puntero del numero de lados adyacentes */
    int ltd = 1; /* lugar de la estructura de datos a almacenar */
    struct cercano *nodo_j; /* puntero de arreglo de nodos adyacentes */
    struct adya *rostro; /* puntero de arreglo de bordes de un elemento */
    unsigned char background; /* imprimir */
    extern int posx = 123;    /* mm */
    extern int posy = 193;     /* mm */



        /* declaracion de funciones */

    int busqueda(int, int, int);
    int nodal(float, float);
    int almacen(int);
    float funcion1(float, float, float, float);
    float funcion2(float, float, float, float);
    float funcion3(float, float, float, float);
    float funcion4(float, float, float, float);
    void error();
    void linea(float, float, float, float);
    void checar();
    void inicia_grafico();
    void almacen2(int, int, int);
    void expande(int);
    void imprimir();
    void resumen();
    char put_out(char);

void etiqueta_asignada()
  /* ---------------------*/
{
    register int i, j, k, v, uv, uu;
    int a=0, num=0, cont1=1;
```

```
struct simple cara[NUM_CARAS];
float x1, x2, y1, y2;
int ros[NUM_CARAS] =
{
3,2,0,2,3,3,0,2,2,0,
2,0,0,0,2,0,1,1,2,3,
0,0,0,0,0,3,0,2,2,3,
0,3,2,0,3,0,0,2,0,2,
2,0,2,0,0,3,3,2,0,2,
2,0
};

inicia_grafico();
nodo   = (float*)malloc(NUM_NODOS*sizeof(float));
nodo_j = (struct cercano*)malloc(NUM_NODOS*sizeof(struct cercano));
nume_1 = (int*)malloc(NUM_NODOS*sizeof(int));
numla  = (int*)malloc(NUM_NODOS*sizeof(int));
prov   = (int*)malloc(10*NUM_NODOS*sizeof(int));
rostro = (struct adya*)malloc(NUM_CARAS*sizeof(struct adya));
if(!nodo || !nodo_j || !nume_1) error();
if(!numla || !prov || !rostro)  error();

for (i=0; i < NUM_CARAS; i++)  /* llena arreglo dinamico */
cara[i+1].nivel = ros[i];

for (i=1; i<=NUM_NODOS; i++)
{
for (j=1; j <= NUM_CARAS; j++)  /* Busca las caras que comparten un */
    {                           /* mismo nodo.                      */
    for (k=1; k <= 4; k++)      /* Busca en los vertices de la      */
    {                           /* cara j.                          */
    if (ELEM(0,j,k) == i)
    {
        PROV(i,num,0) = j; /* Almacena la cara hallada.       */
        PROV(i,num,1) = k; /* Almacena el vertice hallado.    */
         if (a < cara[j].nivel) /* Compara niveles.              */
                a = cara[j].nivel; /* Cambia a un nivel mayor.      */
              num++;
              break;
          }
        }
}

for (k=0; k < num; k++)
    ETIQ(0, PROV(i,k,0), PROV(i,k,1))
          = a;  /* Asigna etiquetas a vertices.  */
NOD(i) = a;
a = 0;
NUM(i) = num; /* numero de caras que comparten al nodo */
num = 0;
}

/* Procedimiento que halla los nodos adyacentes a un nodo */

for (i=1; i <= NUM_CARAS; i++) /* Preproceso */
{                              /* inicializa la estructura */
for (j=1; j <= 4; j++)     /* rostro[i].borde[j][k] */
{                          /* i = superelemento, j = borde */
    for (k=0; k < 2; k++)  /* k = extremo. */
    {
```

```
        if (j+k == 5)
            v = 1;
        else
            v = j+k;
        RBE(i,j,k) = ELEM(0,i,v);
        } /* fin for k */
    } /* fin for j */
    }  /* fin for i */


    for (i=1; i <= NUM_NODOS; i++) /* Busca los nodos adyacentes */
    {
    if (NUM(i) <= 3)
        NLA(i) = NUM(i) + 1;
    else
        NLA(i) = NUM(i);
    x1 = COOR(apu, i,  0);
    y1 = COOR(apu, i,  1);
    for (j=1; j <= NUM_CARAS; j++) /* toma un numero de cara */
    {
        for (k=1; k <= 4; k++)  /* busca en los cuatro bordes */
            {
        uv = RBE(j, k, 0); /* extremo 1 de borde k */
        uw = RBE(j, k, 1); /* extremo 2 de borde k */
        if (uv == i)   /* pregunta si en el extremo 1 esta */
        {               /* el nodo i                        */
            sen = busqueda(uw, cont1-1, i); /* busca si esta almacenado */
                  /* si sen = 0 no ha sido almacenado */
            if (sen == 0)
                    {
            NNA(i, cont1) = uw; /* almacena el nodo */
            x2 = COOR(apu, uw, 0);
            y2 = COOR(apu, uw, 1);
            linea(x1,y1,x2,y2);
            cont1++;
                    }
                }
                else if (uw == i)
                {
            sen = busqueda(uv, cont1, i);
                    if (sen == 0)
                {
            NNA(i, cont1) = uv;
            x2 = COOR(apu, uv, 0);
            y2 = COOR(apu, uv, 1);
            linea(x1,y1,x2,y2);
            cont1++;
                    }
        }
        }
        if (cont1 > NLA(i))
        {
        cont1 = 1;
        break;
        }
        }
    }
    getch();
} /*fin de etiqueta asignada() */
```

```
int busqueda (int numero, int cont1, int i)
/*  ------------ */
{
    register int j, senal=0;

    for (j=1; j <= cont1; j++)
    if (NNA(i,j) == numero)
    {
        senal = 1;
        break;
    }
    return (senal);
}

void extension_admisible()
/*   --------------------- */
{
    int i, j, k, impare=0, pare=0;
    float par[NUM_NODOS], impar[NUM_NODOS];

    for (i=1; i <= NUM_NODOS; i++)
    {
    if (!(i%2))                    /* Si se cumple i es par. */
    {
        if (!NOD(i))
         /* Si la etiqueta asignada es cero entonces */
            {                      /* tomar 0.5.                              */
        par[i-1]   = 0.5; /* Tomar para el grupo par el valor 0.5 */
        impar[i-1] = 0.0; /* Tomar para el grupo impar el valor 0.0 */
            }
        else   /* Si el valor asignado a la etiqueta del nodo i no es cero */
            {
        par[i-1]   = NOD(i); /* Tomar el valor de la etiqueta, en ambos */
        impar[i-1] = NOD(i); /* grupos, asignado al nodo i */
        }
        }
    else    /* Entonces el nodo es impar */
    {
        if (!NOD(i))  /* Si la etiqueta asignada es cero.... */
            {
        impar[i-1] = 0.5; /* para el grupo impar asignar 0.5 */
        par[i-1]   = 0.0; /* y para el impar el valor de cero */
            }
        else  /* Si la etiqueta asignada no es cero..... */
            {
        impar[i-1] = NOD(i); /* a ambos grupos asignar el valor */
        par[i-1]   = NOD(i); /* de la etiqueta del nodo i. */
            }
        }
    }

    for (i=1; i <= NUM_NODOS; i++)
    {
    /* Para el grupo impar: */
    if (impar[i-1] == 0.5)  /* Busca nodos con valor 0.5 */
    {
        for (j=1; j <= NLA(i); j++) /* Busca en los nodos adyacentes... */
            {
        if (impar[NNA(i,j)-1] > 0.5) /* si por lo menos ...*/
        {                   /* en alguno su valor es mayor de 0.5 */
```

```
            impar[i-1] = 1;  /* Hallo un valor mayor a 0.5 */
            break;  /* cambiar al siguiente nodo */
                }
        else if (j == NLA(i))  /* sino encontro un valor mayor de 0.5 */
        {    /* en los nodos adyacentes..... */
            impar[i-1] = 0; /* asignar el valor de cero a este */
            impare++;  /* ademas llevar su registro. */
                }
            }
    }
    /* Para el grupo par: */
    if (par[i-1] == 0.5)  /* Busca la etiqueta 0.5 en los nodos */
        {
        for (j=1; j <= NLA(i); j++)
            {
        if (par[NNA(i,j)-1] > 0.5) /* Revisa en nodos */
                {                                        /* adyacentes.      */
            par[i-1] = 1; /*encontro un valor mayor a 0.5 asignar un uno*/
            break; /* cambiar al siguiente nodo */
                }
        else if (j == NLA(i)) /* sino encontro un valor mayor a 0.5 en los  */
        {   /* nodos adyacentes..... */
            par[i-1] = 0; /* asignar el valor de cero */
            pare++;  /* llevar su registro */
                }
            }
    }
    }
    if (pare > impare)  /* comparar los registros de ceros y */
    {                   /* tomar el grupo que registre mas ceros */
    for (i=1; i < NUM_NODOS; i++)
        for (k=1; k <= NUM(i); k++) /* el grupo par es escogido */
         ETIQ(apu, PROV(i,k,0), PROV(i,k,1)) = par[i-1];
    }
    else  /* si el registro de ceros del grupo impar es mayor.... */
    {
    for (i=1; i <= NUM_NODOS; i++)  /* este es escogido */
        for (k=0; k < NUM(i); k++)
         ETIQ(apu, PROV(i,k,0), PROV(i,k,1)) = impar[i-1];
    }
    free(node);   /* libera bloques de memoria */
    free(nume_1);
    free(rostro);
    free(numla);
    free(nodo_j);
} /* fin de etiqueta asignada */

void subdivide_2()
/*   ------------- */
{
    register int i, j, c, v;
    int vac=1, contador=0;
    int it, rs;
    int k, ab = 100;
    float m[2], b[2];

    for (i=1; i < 5; i++)  /* Calcula etiquetas de esquinas */
    {
        etiqueta[i] = etiqueta[i] - 1;
        if (etiqueta[i] < 0)
```

```
        etiqueta[i] = 0;
    }

    for (i=1; i <= 4; i++) /* Calcula etiquetas de puntos medios */
    {
    it = i%4+1;
    if (etiqueta[i] < etiqueta[it])  /* compara las etiquetas de los */
        etiqueta[i+4] = etiqueta[i];/* extremos de un borde y toma  */
    else                            /* el valor menor.              */
        etiqueta[i+4] = etiqueta[it];
    }

    /* Calculando la etiqueta central.
       Tomara el valor minimo de las etiquetas medias,
       calculado en el paso anterior, como el valor
       de la etiqueta central. Este valor sera cero
       unicamente si todas las etiquetas medias son cero */
    for (i=5; i < 9; i++)
    {
        if (etiqueta[i] == 0)
            contador++;
    else if (etiqueta[i] < ab)
            ab = etiqueta[i];
    }
    if (contador == 4)
        etiqueta[9] = 0;
    else
        etiqueta[9] = ab;

    for (i=1; i <= 4; i++)   /* Calcula coordenadas de puntos medios */
        for (j=0; j < 2; j++)
    {
        rs = i%4+1;
        coord[i+4][j] = (coord[rs][j] - coord[i][j])/2
                + coord[i][j];
    }


    for (i=5; i < 7; i++)
    {                   /* Calcula coordenadas centrales */
    b[i-5] = funcion1(coord[i][0], coord[i][1],
                      coord[2+i][0], coord[2+i][1]);
    m[i-5] = funcion2(coord[i][0], coord[i][1],
                      coord[2+i][0], coord[2+i][1]);
    }
    coord[9][0] = funcion3(b[0],m[0],b[1],m[1]);
    coord[9][1] = funcion4(b[0],m[0],b[1],m[1]);
/*    printf("centrales x=%f y=%f ", coord[9][0], coord[9][1]);
  */
    for (k=5; k<9; k++)
     linea(coord[9][0],coord[9][1],
           coord[k][0],coord[k][1]);  /* traza lineas en elemento */


    expande(4); /* expande memoria dinamica */

    for (c=1; c  <= 4; c++) /* almacena los datos calculados */
    {
    v = vac++;
    almacen2(c, c, c);    /* almacena etiqueta y nodo */
```

```
    v = v%4+1;
    almacen2(c, v, c+4);  /* almacena etiqueta y nodo */
    v = v%4+1;
    almacen2(c, v, 9);    /* almacena etiqueta y nodo */
    v = v%4+1;
    almacen2(c, v, v+4);  /* almacena etiqueta y nodo */
    }
    numero_rostros+= 4;
} /* fin de subdivide2() */

void subdivide_paralelo()
/*   -------------------  */
{
    register int i, j, k;
    int rostro1;
    int igual, contar=0, contador1;
    void subdivide_1 (int, int);
    void subdivide_2(void);
    int qp;
    char cla_ve;

    cla_ve = getch();  /* el proceso fué completado */
    if(cla_ve == 'i') Ld_Graphic(0);   /* imprimir(); */

/*    imprimir();   imprimir en EPSON */
    do
    {
    for (i=1; i <= numero_caras; i++)
    {
        contador1 = 0;
        for (j=1; j<=4; j++)
        {
        qp = ELEM(apu, i, j); /* Almacena al nodo
                 del vertice de la cara */
        etiqueta[j] = ETIQ(apu, i, j); /* Almacenamiento local */
                        /* de etiquetas.*/
        for (k=0; k < 2; k++)  /* Almacena las coordenadas del nodo */
            coord[j][k] = COOR(apu, qp, k);
        if (!etiqueta[j])
            contador1++; /* Evalua el numero de ceros en las etiquetas. */
        else   /* Almacena los valores de la ultima etiqueta diferente */
        {      /* de cero. */
            igual   = j;   /* vertice */
            rostro1 = i;   /* superelemento */
        }
        } /* fin de if j */
        if (contador1 == 3)  /* Si una sola etiqueta es diferente de cero... */
        subdivide_1(igual,rostro1); /* realizar subdivide_1. */
        else if (contador1 == 4) /* Si todas las etiquetas son cero... */
        contar++;  /* el elemento ha sido completado, llevar su registro. */
        else
        subdivide_2();

    } /* fin de if i */
/*  closegraph();
    checar(); */
/*  imprimir();   imprimir proceso de mallado */
    if (contar != numero_caras) /* Si en los superelementos no se */
    {                           /* llevo acabo el proceso de subdivision */
        numero_caras = numero_rostros; /* Cambia al nuevo grupo de caras */
```

```
        contador1 = 0;  /* inicializa las variables */
        contar    = 0;
        numero_nodos1 = numero_nodos2; /* Cambia al nuevo grupo de nodos */
        numero_rostros = 0;
        numero_nodos2  = 1;
        apu = almacen(apu);
        itd = almacen(apu); /* Cambia a un nivel de almacenaje general */
    } /* fin de if */
    else
    {
        cla_ve = getch();  /* el proceso fué completado */
        if(cla_ve == 'i') LJ_Graphic(0);   /* imprimir(); */
        closegraph();
        resumen();      /* elementos y nodos generados */
        exit(0);   /* salir */
    }
    } while(1);
} /* fin de subdivide_paralelo() */

void subdivide_1(int kapa, int ros)
/*   -------------  */
{
    struct orden
    {
    int i[4];
    } graber[3] = {  1, 5, 7, 6 ,  5, 2, 3, 7 ,  6, 7, 3, 4  };

    int i, j, nod[5];
    float m[2], b[2];
    int lt, rs;
    int parc;

    etiqueta[1] = etiqueta[kapa];
    for (i=1; i < 5; i++)            /* Cambia coeficientes */
    {
    nod[i]      = ELEM(apu, ros, kapa);
    coord[i][0] = COOR(apu, nod[i], 0);
    coord[i][1] = COOR(apu, nod[i], 1);
    kapa = kapa%4+1;
    }

    for (i=1; i <= 9; i++) /* Calcula etiquetas */
    {
        if (i == 1)
            etiqueta[i] = etiqueta[i]-1;
        else
            etiqueta[i] = 0;
    }
    for (i=1; i <= 4; i++)   /* Calcula coordenadas de puntos medios */
        for (j=0; j < 2; j++)
    {
        rs =i%4+1;
        coord[i+4][j] = (coord[rs][j] - coord[i][j])/2
                + coord[i][j];
    }

    for (i=5; i < 7; i++)
    {                   /* Calcula coordenadas centrales */
    b[i-5] = funcion1(coord[i][0], coord[i][1],
                      coord[2*i][0], coord[2*i][1]);
```

```
    m[i-5] = funcion2(coord[i][0], coord[i][1],
                      coord[2+i][0], coord[2+i][1]);
    }
    coord[7][0] = funcion3(b[0],m[0],b[1],m[1]);
    coord[7][1] = funcion4(b[0],m[0],b[1],m[1]);
    coord[6][0] = coord[8][0];
    coord[6][1] = coord[8][1];
    linea(coord[7][0],coord[7][1],
      coord[3][0],coord[3][1]);
    linea(coord[7][0],coord[7][1],
      coord[5][0],coord[5][1]);
    linea(coord[7][0],coord[7][1],
      coord[6][0],coord[6][1]);

    for (i=1; i < 8; i++)
    for (j=0; j < 2; j++)
    {
/*      printf("coord[%d][%d]=%f ", i, j, coord[i][j]);
*/
    }
    expande(3); /* expande memoria dinamica */

    for (i=1; i < 4; i++)   /* Almacena los valores calculados. */
        for (j=1; j < 5; j++)
    {
        parc = grabar[i-1].i[j-1];
        ETIQ(ltd, numero_rostros+i, j) =
             etiquetas[parc]; /* Etiquetas */
        ELEM(ltd, numero_rostros+i, j) =
             nodal(coord[parc][0],
                   coord[parc][1]); /* Nodos */
        }
    numero_rostros+=3; /* se generaron tres elementos */
}


float funcion1 (float x1, float y1, float x2, float y2)
/*   ------------    */
{
    float cero;

    cero = x2 - x1;
    if (cero == 0)
    cero = 0.00000001;
    return((y1*x2-y2*x1)/(cero));
}

float funcion2 (float x1, float y1, float x2, float y2)
/*    ----------    */
{
    float cero;

    cero = x2 - x1;
    if (cero == 0)
    cero = 0.00000001;
    return((y2-y1)/(cero));
}

float funcion3 (float b1, float m1, float b2, float m2)
/*    ---------    */
```

```
{
    return((b2-b1)/(m1-m2));
}

float funcion4 (float b1, float m1, float b2, float m2)
/*  ----------  */
{
    return((b2*m1-b1*m2)/(m1-m2));
}
int nodal(float x, float y)
/*  -------  */
{
    register int i, nudo, dice=0;

    for (i=1; i < numero_nodos2; i++)
    {
    if (x == COOR(itd, i, 0))
          if (y == COOR(itd, i, 1))
          {
          nudo = i; /* Hallo un nodo con las mismas coordenadas */
          dice = 1;
          break;
          }
    }
    if (!dice)
    {
    nudo = numero_nodos2++; /* No hallo algun nodo con estas coordenadas */
    proc[itd].coordenadas.nodo =
            realloc(proc[itd].coordenadas.nodo,
            numero_nodos2*sizeof(struct doe));
    if(!proc[itd].coordenadas.nodo) error();
    COOR(itd, nudo, 0) = x;
    COOR(itd, nudo, 1) = y;

    }

/*    printf(" proc[%d].coordenadas.nodo[%d].xy[0]= %f ", itd,
        nudo, proc[itd].coordenadas.nodo[nudo].xy[0]);
    printf(" proc[%d].coordenadas.nodo[%d].xy[1]= %f ", itd,
        nudo, proc[itd].coordenadas.nodo[nudo].xy[1]);
    */
    return(nudo);
}

void leer(void)
/*  -------  */
{
    register int i,j;
    int count = 0;


/* Coordenadas de cada nodo de los superelementos */
    float coor[NUM_NODOS][2] = {
    1.1,1.6,2.7,1.6,2.7,2.8,2.2,3.0,4.3,1.6,3.1,3.0,3.3,3.4,
    4.3,3.2,4.3,4.6,3.1,3.9,2.7,4.1,2.7,5.5,1.1,4.7,2.2,3.9,
    2.0,3.4,1.1,3.2,0.0,3.0,0.0,0.0,2.7,0.0,5.3,0.0,5.3,3.0,
    5.3,5.4,3.5,6.6,0.0,5.2,0.0,7.7,4.4,7.7,8.7,7.7,8.7,3.9,
    8.7,0.0,13.1,0.0,17.4,0.0,17.4,3.9,13.1,3.9,13.1,7.7,
    17.4,7 7,17.4,13.1,15.0,14.1,14.7,10.6,11.9,10.0,8.7,9.8,
    5.6,10.0,2.8,10.6,2.4,14 1,0.0,13.1,0.0,18.5,2.4,17.7,
```

```
    3.7,20.0,2.1,21.5,5.0,23.6,5.9,21.6,8.7,22.3,8.7,24.5,
    12.4,23.6,11.5,21.6,13.7,20.0,15.4,21.5,17.4,18.5,15.0,17.7,
    12.6,16.9,12.6,15.1,11.9,13.5,10.5,12.5,8.7,12.1,6.9,12.5,
    5.5,13.5,4.8,15.2,4.8,16.9,5.6,18.6,6.7,19.6,8.7,20.1,
    10.6,19.6,11.8,18.6
                    );

/* Nodos de cada superelemento */

    int nod[NUM_CARAS*4] = {
    1,2,3,4,5,6,3,2,5,8,7,6,9,10,7,8,9,10,11,12,11,12,13,14,
    15,14,13,16,15,16,1,4,19,2,1,18,19,20,5,2,21,8,5,20,21,22,9,8,
    9,22,23,12,13,12,23,24,17,16,13,24,1,16,17,18,29,28,21,20,
    21,28,27,22,27,26,23,22,23,26,25,24,29,30,33,28,33,34,27,28,
    31,32,33,30,33,32,35,34,25,26,41,42,27,40,41,26,39,40,27,34,
    35,38,39,34,25,42,43,44,65,66,43,42,41,64,65,42,41,40,63,64,
    39,62,63,40,39,38,61,62,37,60,61,38,35,36,37,38,43,46,45,44,
    43,66,67,46,37,58,59,60,37,36,57,58,45,46,47,48,67,68,47,46,
    59,58,55,72,57,56,55,58,47,50,49,48,69,50,47,68,71,72,55,54,
    55,56,53,54,69,70,51,50,51,52,49,50,71,54,51,70,51,52,53,54
                    };

     /* inicializa memoria dinamica */
    for (i=0; i<=1; i++)
    {
    proc[i].etiquetas.cara   = (struct cuatro*) malloc
                    (NUM_CARAS*sizeof(struct cuatro));
    proc[i].coordenadas.nodo = (struct dos*) malloc
                    (NUM_NODOS*sizeof(struct dos));
    proc[i].nodos.elemento   = (struct cuatro*) malloc
                    (NUM_CARAS*sizeof(struct cuatro));
    if(!proc[i].etiquetas.cara)   error();
    if(!proc[i].coordenadas.nodo) error();
    if(!proc[i].nodos.elemento)   error();
    }

    for(i=1; i<=NUM_CARAS; i++)
    for(j=1; j<=4; j++)
    {
        ELEM(0, i, j) = nod[count];
        count++;
    }
    for(i=1; i<=NUM_NODOS; i++)
     for (j=0; j<2; j++)
         COOR(0, i, j) =  coor[i-1][j];
} /* fin de funcion leer() */

int almacen(int w)
/*  --------  */
{
    if (w == 0)
    w = 1;
    else
    w = 0;
    return (w);
}

void error()
/*  --------  */
{
```

```
    printf(" Capacidad de memoria excedida \n ");
    printf(" memoria disponible %ld bytes \n ", coreleft());
    getch();
    closegraph();
    exit(1);
}

void checar()
/*  ..........  */
{
    register int i, j, count;

    count = 0;
    for(i=1; i<=numero_nodos2; i++)
    for(j=0; j<2; j++)
    {
        count++;
        printf("\n coor(%d,%d)=%f ",i,j,COOR(1,i,j));
        if(count==24)
        {
        getch();
        count=0;
        }
    }
    count=0;
    for(i=1; i<=numero_rostros; i++)
    for(j=1; j<=4; j++)
    {
        count++;
        printf("\n elem(%d,%d)=%d ",i,j, ELEM(1,i,j));
        if(count==24)
        {
        getch();
        count=0;
        }
    }
 /*   inicia_grafico();*/
} /* fin de checar() */

void linea(float x1, float y1, float x2, float y2)
/* ...........  */
{
    int car;

    line(x1*ESC_X+10, 450-ESC_Y*y1,
     x2*ESC_X+10, 450-ESC_Y*y2);
    if(kbhit())
    {
    car = getch();
    if(car == 's')
    {
        closegraph();
        exit(0);
    }
    }
}

void inicia_grafico()
/*  ... .. ........  */
{
```

```
    int g_driver, g_mode, g_error; /* Para definir el tipo de dispositivo */
    /* inicializa el dispositivo grafico */
    detectgraph(&g_driver, &g_mode);
    initgraph(&g_driver, &g_mode, "c:\\tc");
    g_error = graphresult();
    if(g_error < 0)
    {
    printf("ERROR:%s.\n", grapherrormsg(g_error));
    getch();
    exit(1);
    }
    cleardevice();    /* limpiar el dispositivo */
    setbkcolor(LIGHTBLUE);
    line(3, 3, 635, 3);
    line(3, 3, 3, 475);
    line(635, 475, 635, 3);
    line(635, 475, 3, 475);
}

/* La siguiente rutina almacena los datos de cada elemento */
void almacen2(int cara_p, int vert_p1, int vert_p2)
/*   ---------  */
{
    cara_p += numero_rostros;
    ETIQ(ltd, cara_p, vert_p1) =
     etiqueta[vert_p2];              /* etiquetas */
    ELEM(ltd, cara_p, vert_p1) =
     nodal(coord[vert_p2][0],
           coord[vert_p2][1]);       /* nodos */
}

void expande(int canti)
{
          /* Expande memoria dinamica */

    proc[ltd].nodos.elemento = (struct cuatro*)realloc
         (proc[ltd].nodos.elemento,((numero_rostros
             +canti)*sizeof(struct cuatro)));
    proc[ltd].etiquetas.cara =(struct cuatro*)realloc
         (proc[ltd].etiquetas.cara,((numero_rostros
             +canti)*sizeof(struct cuatro)));
    if(!proc[ltd].nodos.elemento) error();
    if(!proc[ltd].etiquetas.cara) error();
}

void imprimir()
{
/*   char put_out(char);*/
    int i, x, y;

    put_out(0x1b);
    put_out(0x33);
    put_out(0x16);
    for(i=0; i<15; i++)
    put_out(0x0A);    /* LF salto de renglon */
    background = readPixel(0,0);
    for(x=647; x>0; x-=8)
    {
    if(kbhit())
    {
```

```
        for(i=1; i<=2; i++)
        printf("%c",CAMP);
        put_out(0x1b);
        put_out(0x32);
        getch();
        break;
    }
    printrow(x);
    }
    put_out('\x0C'); /* avance de pagina  FF */
}

printrow(int x)
{
    static unsigned char savechar1, savechar2, savechar3,
          temp;
    static unsigned char out_buff[1684]={"\x1bZ\x91\x06"};
    unsigned int i, j, newy, y;
    char status();

    for(y=0, j=239; y<=479; y++, j+=3)
    {
    savechar1 = 0;
    savechar2 = 0;
    savechar3 = 0;
    for(i=0; i<8; i++)
    {
        temp = readPixel(x-i, y);
        if(temp != background)
        {
        if((temp & 0x01) != background)
            savechar1 |= 1;
        if((temp & 0x02) != background)
            savechar2 |= 1;
        if((temp & 0x04) != background)
            savechar3 |= 1;
        }
        if(i != 7)
        {
        savechar1 <<= 1;
        savechar2 <<= 1;
        savechar3 <<= 1;
        }
    }
    out_buff[j]   = savechar1;
    out_buff[j+1] = savechar2;
    out_buff[j+2] = savechar3;
    }
    for(i=0; i<239; i++)
    put_out(out_buff[i]);
    for(i=239; i<=1684; i+=2)
    {
    put_out(out_buff[i]);
    put_out(0x00);
    }
    put_out('\r');
    for(i=0; i<239; i++)
    put_out(out_buff[i]);
    for(i=240; i<=1684; i+=2)
    {
```

```
    put_out(0x00);
    put_out(out_buff[i]);
    }
    put_out('\r');
    put_out('\n');
}/* fin de printrow */

char put_out(char character)
{
    union REGS reg;

    while(!status());
    reg.h.ah = 0;
    reg.h.al = character;
    reg.x.dx = 0;
    int86(0x17, &reg, &reg);
    return(reg.h.ah);
 }

char status()
{
    union REGS reg;

    reg.h.ah = 2;
    reg.x.dx = 0;
    int86(0x17, &reg, &reg);
    return(reg.h.ah & 0x80);
}

int readPixel(int x, int y)
{
    union REGS reg;

    reg.h.ah = 0x00;
    reg.x.cx = x;
    reg.x.dx = y;
    int86(0x10, &reg, &reg);
    return(reg.h.al);
}

void resumen()
{

    printf("\n ********** RESUMEN **********");
    printf("\n Numero de nodos     %d", numero_nodos1);
    printf("\n Numero de elementos %d", numero_caras);
}

void main()
{
    void leer(void);
    void etiqueta_asignada(void);
    void extension_admisible(void);
    void subdivide_paralelo(void);

    leer();
    etiqueta_asignada();
    extension_admisible();
    subdivide_paralelo();
    getch();
```

```
    closegraph();
    resumen();
}
```

```
/* Complemento del programa generador de elementos triangulares*/

#include <stdio.h>
#include <alloc.h>
#include <graphics.h>
#include <math.h>
#include <dos.h>
#include <conio.h>
#include "util.h"
int transferencia(int, int);
float dist(int, int);
void inic(int);
float cuadrante(float, float);
void error();
float p_dis(int);
float ang_mx(int, int);
void reordenar1(int*);
void pool();
float angulo(int, int, int);
float *rotar(float, float, float, float, int);
void refinado(int, int);
void lineas(int, int);
void imprimir();
char background;
void limpiar();     /* Limpia de antiguos trazos */
void resumen();
void almacen(int, int, int);
int busqueda(float, float);
int busqueda(float, float);

float dist(int nodo_1, int nodo_2)
  /*--------*/
{
    float x0, y0, x1, y1, distancia;

    x0 = COORD(nodo_1, 1);
    y0 = COORD(nodo_1, 2);
    x1 = COORD(nodo_2, 1);
    y1 = COORD(nodo_2, 2);
    distancia = sqrt( pow((x1-x0),2) + pow((y1-y0),2) );
    return(distancia);
} /* fin de la funcion dist */

/* La siguiente rutina calcula la distancia promedio de un nodo con
   sus nodos adyacentes        */
float p_dis(int nod)
  /*-------*/
{
    register int i;
    int noda=0;
    float dist_p = 0;

    for(i=1; i<=NNA(nod); i++)
    {
    noda = NDNA(nod, i);
    dist_p += dist(noda, nod);
    }
    return(dist_p/NNA(nod));
} /* fin de p_dis() */
```

```
void error()
  /*---- ---*/
{
    printf("Capacidad de memoria excedida\n");
    printf("Memoria disponible %ld bytes\n", coreleft());
    getch();
    closegraph();
    resumen();
    exit(1);
}

void inic(int nudo)
 /*-------*/
{                   /* inicializa el puntero de nodos adyacentes */
    PN(nudo) = (nodos+nudo-1)->nodosa = DEFN3(2);
    PN(nudo)++; /*cambia ala posicion siguiente */
}

 /* La siguiente funcion determina en que
 cuadrante se encuentran las coordenadas */

float cuadrante(float x, float y)
  /*------------*/
{
    float cuadra;

    if(x < 0)
    if(y < 0)
        cuadra = 180;      /* tercer cuadrante */
    else
        cuadra = 180;      /* segundo cuadrante */
    else
    if(y < 0)
        cuadra = 360;      /* cuarto cuadrante */
    else
        cuadra =   0;      /* primer cuadrante */
    return(cuadra);
} /* fin de funcion cuadrante */

int transferencia(int nodo_t, int nivel)
 /*--------------*/
{
    int nodal;

    nodal = nodo_t;
    if(nivel > 0)
    {
    nodal = *PN(nodo_t);
    nivel--;
    if(nivel)
        nodal = transferencia(nodal, nivel);
    }
    else if(nivel < 0)
    {
    nodal = *( PN(nodo_t) - 1 );
    nivel++;
    if(nivel < 0)
    nodal = transferencia(nodal  nivel);
    }
    return(nodal)
```

```
}


float ang_mx(nodo1, nodo2)
   /*--------*/
{
    register int i, j;
    int nodal[2], nodo[2];
    float ang_par, ang_men=0;

    nodo[0] = nodo1;
    nodo[1] = nodo2;
    for(i=0; i<2; i++)
    {
    nodal[0] = transferencia(nodo[i], -1);  /* nodo adyacente a i */
    nodal[1] = transferencia(nodo[(i+1)%2], 1); /* nodo adyacente */
    ang_par = angulo(nodal[0], nodo[i],
              nodo[(i+1)%2]); /* primer sentido */
    if(!ang_men) ang_men = ang_par;
    if(ang_par < ang_men) ang_men = ang_par;
    ang_par = angulo(nodo[i], nodo[(i+1)%2],
              nodal[1]); /* segundo sentido */
    if(ang_par < ang_men) ang_men = ang_par;

    }
    return(ang_men);
 } /* fin de funcion ang_mx */

 void reordenar1(int *nod)
   /*------------*/
 {
    register int i;
    int n;
    int ini_1, ini_2;
    int fin_1, fin_2;
    float direc1, direc2;

    ini_1 = *(nod+2);
    ini_2 = *(nod+3);
    fin_1 = *nod;
    fin_2 = *(nod+1);
    cambio(ini_2, ini_1);
    cambio(fin_1, fin_2);
    direc1  = angulo(ini_2, ini_1, fin_1);
    direc1 += ang1;
    n = ini_1;
    for(;;)
    {
    for(i=1; i<=NNA(n); i++)
    {
        direc2 = angulo(ini_2, ini_1, NDNA(n,i));
        direc2 += ang1;
        if(direc2 >= direc1-1 && direc2 <= direc1+1)
        {   /* evita el regreso al nodo anterior */
        if(*(PN(n)-1) == NDNA(n,i)) continue;
        cambio(n, NDNA(n, i));  /* aceptado el cambio */
        n = *PN(n)
        break;
        }
```

```
    if(n == fin_1) break;
    }
} /* fin de la funcion reordenar1() */

void pool()
 /*----------*/
{
    long maximo;

    maximo = coreleft();
    printf("\n tamaño maximo del pool = %ld bytes\n", maximo);
    getch();
}

float *rotar(float x1, float y1, float x2, float y2, int sent)
{
    float co_rt[2], xm, ym;

    xm = x2 - x1;
    ym = y2 - y1;
    co_rt[0]  = xm*cos(PI/3*sent) - ym*sin(PI/3*sent);
    co_rt[1]  = xm*sin(PI/3*sent) + ym*cos(PI/3*sent);
    co_rt[0] += x1;
    co_rt[1] += y1;
    return(&co_rt[0]);
}
void refinado(int n_central, int n_lateral)
{
    int nodo1, nodo2;
    float xx, yy;
    float *prt_rt;
    float mediax=0, mediay=0;
    register int i;
    float x1, y1, x2, y2;
    int n_ve, n_no, le, ka;
    float ab, bc;
    int sitio;

    if(n_central <= parametro) return;
    for(i=1; i<=n_co; i++)
    {          /* si es un nodo de un conector no lo mueve */
    if(n_central >= CONEC(i,2) && n_central <= CONEC(i,4))
        return;
    if(n_central == CONEC(i,1) || n_central == CONEC(i,3))
        return;
    }
    n_ve = NEA(n_central);
    n_no = NNA(n_central);
    setcolor(LIGHTRED);

    for(i=1; i<=n_no; i++)
    {
    linea(n_central, NONA(n_central,i));/* borra las anteriores divisiones */
    }
    for(i=1; i<=n_no; i++)   /* busca el lugar que ocupa */
    if(NONA(n_central,i) == n_lateral)
    {
        sitio = i-1;
        break;
    }
```

```
    for(i=1; i<=n_ve; i++)
    {
    ka = sitio%n_no+1;                  /* primer coeficiente */
    le = (sitio+1)%n_no+1;              /* segundo coeficiente */
    nodo1 = NDNA(n_central, ka);        /* primer nodo */
    nodo2 = NDNA(n_central, le);        /* segundo nodo */
    x1 = COORD(nodo1, 1);
    y1 = COORD(nodo1, 2);
    x2 = COORD(nodo2, 1);
    y2 = COORD(nodo2, 2);
    prt_rt  = rotar(x1, y1, x2, y2, 1);
    ab = *prt_rt;
    bc = *(prt_rt+1);
    mediax += ab;
    mediay += bc;
    sitio++;
    }
    setcolor(15);
    mediax = mediax/n_ve;
    mediay = mediay/n_ve;
    COORD(n_central, 1) = mediax;
    COORD(n_central, 2) = mediay;
    mediax = X(mediax);
    mediay = Y(mediay);
    for(i=1; i<=n_no; i++)
    {
    xx = COORD(NDNA(n_central, i), 1);
    yy = COORD(NDNA(n_central, i), 2);
    line(mediax, mediay, X(xx), Y(yy));
    }
} /* fin de funcion refinado */


/* traza una linea de nod_1 a nod_2 */
void linea(int nod_1, int nod_2)
{
    float x1, y1, x2, y2;

    x1 = COORD(nod_1, 1);
    y1 = COORD(nod_1, 2);
    x2 = COORD(nod_2, 1);
    y2 = COORD(nod_2, 2);
    x1 = X(x1);
    y1 = Y(y1);
    x2 = X(x2);
    y2 = Y(y2);
    line(x1, y1, x2, y2);
}

void imprimir()
{
    char put_out(char);
    int i, x, y;

    put_out(0x1b);
    put_out(0x33);
    put_out(0x16);
    for(i=0; i<12; i++)
    put_out(0x0A);    /* LF salto de renglon */
```

```
    background = readPixel(0,0);
    for(x=639; x>7; x-=8)
    {
    if(kbhit())
    {
        put_out(0x1b);
        put_out(0x32);
        getch();
        break;
    }
    printrow(x);
    }
    put_out('\x0C'); /* avance de pagina  FF */
}

printrow(int x)
{
    unsigned char savechar1, savechar2, savechar3,
          temp;
    static unsigned char out_buff[1534]={"\x1bZ\xFA\x05"};
    unsigned int i, j, newy, y;
    char status();

    for(y=0, j=89; y<=479; y++, j+=3)
    {
    savechar1 = 0;
    savechar2 = 0;
    savechar3 = 0;
/*  savechar4 = 0;*/
    for(i=0; i<8; i++)
    {
        temp = readPixel(x-i, y);
        if(temp != background)
        {
        if((temp & 0x01) != background)
            savechar1 |= 1;
        if((temp & 0x02) != background)
            savechar2 |= 1;
        if((temp & 0x04) != background)
            savechar3 |= 1;
/*      if((temp & 0x08) != background)
            savechar4 |= 1; */
        }
        if(i != 7)
        {
        savechar1 <<= 1;
        savechar2 <<= 1;
        savechar3 <<= 1;
/*      savechar4 <<= 1; */
        }
    }
    out_buff[j]   = savechar1;
    out_buff[j+1] = savechar2;
    out_buff[j+2] = savechar3;
/*  out_buff[j+3] = savechar4;*/
    }
    for(i=0; i<89; i++)
    put_out(out_buff[i]);
    for(i=89; i<1534; i+=2)
    {
```

```
    put_out(out_buff[i]);
    put_out(0x00);
    }
    put_out('\r');
    for(i=0; i<89; i++)
    put_out(out_buff[i]);
    for(i=90; i<1534; i+=2)
    {
    put_out(0x00);
    put_out(out_buff[i]);
    }
    put_out('\r');
    put_out('\n');
}/* fin de printrow */

char put_out(char character)
{
    union REGS reg;

    while(!status());
    reg.h.ah = 0;
    reg.h.al = character;
    reg.x.dx = 0;
    int86(0x17, &reg, &reg);
    return(reg.h.ah);
}

char status()
{
    union REGS reg;

    reg.h.ah = 2;
    reg.x.dx = 0;
    int86(0x17, &reg, &reg);
    return(reg.h.ah & 0x80);
}

int readPixel(int x, int y)
{
    union REGS reg;

    reg.h.ah = 0x0D;
    reg.x.cx = x;
    reg.x.dx = y;
    int86(0x10, &reg, &reg);
    return(reg.h.al);
}

/* la siguiente rutina limpia el
   dispositivo de lineas anteriores   */
void limpiar()
{
    register int i,j;

    cleardevice();
    for(i=1; i<=nnodosg; i++)
    for(j=1; j<=NNA(i); j++)
        linea(i, NDNA(i,j));
}
```

```
void resumen()
{
    printf("\n**************** RESUMEN ****************");
    printf("\nNumero de elementos generados: %d\n", nelementosg);
    printf("Numero de nodos generados: %d\n"  nnodosg);
    getch();
}

void almacen(int nd1, int nd2, int nd3)
 /*---------*/
{
    nelementosg++;
    ielemento = DEF44(nelementosg);
    ELNA(nelementosg, 1) = nd1;  /* nodo uno del elemento */
    ELNA(nelementosg, 2) = nd2;  /* nodo dos */
    ELNA(nelementosg, 3) = nd3;  /* nodo tres */
    inic_elem(nd1); /* incremente e inicializa memoria dinamica */
    inic_elem(nd2);
    inic_elem(nd3);
    NDEA(nd1, NEA(nd1)) = nelementosg;  /* almacena nodo adyacente */

    NDEA(nd2, NEA(nd2)) = nelementosg;
    NDEA(nd3, NEA(nd3)) = nelementosg;
} /* fin de funcion almacen() */

/* Esta rutina tiene como objetivo conocer si el nodo ya ha
   sido generado */
int busqueda(float xbus, float ybus)
 /*-----------*/
{
    register int i;
    int almac = -1;
    float xxx, yyy;

    for(i=1; i<=nnodosg; i++)
    {
      xxx = COORD(i,1);
      yyy = COORD(i,2);
    if(xxx == xbus  &&  yyy == ybus)
    {
        almac = i;   /* encontro un nodo con estas coordenadas */
        return(almac);
    }
    }
    return(almac);  /* no encontro un nodo con estas coordenadas */
}
```

```
/* progama maneja.c
   Controla la impresión de la figura en pantalla */

#define VERTICAL   0
#define HORIZONTAL 1
#include <graphics.h>
#include <stdio.h>
extern int posx, posy;     /* variable global */

void Print_Graph(int Modo, int Direccion)
{
    char m;
    int i, j, k, Msb, Lsb,
    MaxX = getmaxx(),
    MaxY = getmaxy();

    setviewport(0, 0, MaxX, MaxY, 0);
    fprintf(stdprn,"\x1b\x4A%c", 7);
    switch(Direccion)
    {
    case VERTICAL:
    {
        Lsb = MaxX & 0x00FF;
        Msb = MaxX >> 8;
        for(j=0; j<=MaxY/8; j++)
        {
        fprintf(stdprn, "\x18*%c%c%c", Modo, Lsb, Msb);
        for(i=0; i<=MaxX; i++)
        {
            m = 0;
            for(k=0; k<8; k++)
            {
            m<<=1;
            if(getpixel(i,j*8+k)) m++;
            }
            fprintf(stdprn,"%c", m);
        }
        fprintf(stdprn,"\x0D\x0A");
        }
    }
    case HORIZONTAL:
    {
        Lsb = MaxY & 0x00FF;
        Msb = MaxY >> 8;
        for(j=0; j<MaxX; j+=8)
        {
        fprintf(stdprn,"\x18*%c%c%c", Modo, Lsb, Msb);
        for(i=MaxY; i>=0; i--)
        {
            m = 0;
            for(k=0; k<8; k++)
            {
            m<<=1;
            if(getpixel(j+k,i)) m++;
            }
            put_out(m);
            /* f(stdprn,"%c", m);*/
        }
        fprintf(stdprn,"\x0D\x0A");
        }
```

```
    }
    fprintf(stdprn "\f")
}/* fin de Print_Graph() */

/* La siguiente subrutina tiene por objetivo
   direccionar hacia la impresora Laser-Jet  */

void LJ_Graphic(int Modo)
{
    int i, j, k, p, q, xasp, yasp,
    MaxX = getmaxx()+1,
    MaxY = getmaxy()+1;
    static char graph_fin[] = "  \x1B*r8";
    static char graph_inic[] =
        "\x1B*E\x1B&l1H\x1B&l00\x1B*p0X\x1B*p0Y\x1B*t";
    double xprint, yprint, prstep, AspR;
    char  m, resolucion[3], temp, m2, back;

    getaspectratio(&xasp, &yasp);
    AspR = (double)xasp/(double)yasp;
    setviewport(0, 0, MaxX, MaxY, 0);
    switch(Modo)
    {
    case VERTICAL:
    {
        xprint = 28.37*posx   284;      /* 1300 */
        yprint = 28.41*posy   369;    /* 600.0, 3900 */
        strcpy(resolucion,"150");
        prstep = 4.8/AspR;    /* 4.8, 2.4 */
        fprintf(stdprn, "%s%s%s",graph_inic, resolucion);
        back = getpixel(0,0);
        for(j=0; j<MaxY; j++)
        {
            fprintf(stdprn, "\x1B&a%-*.1fh%-*.1fV ",
                format(xprint), xprint,
                format(yprint), yprint );
            yprint+=prstep;
            fprintf(stdprn,"\x1B*r1A\x1B*b%dW", MaxX/8);
            for(i=0 ; i<MaxX/8; i++)
            {
            m = 0;
            m2 = 1;
            for(k=0; k<8; k++)
            {
                m2 <<= 1;
                m<<=1;
                temp = getpixel(i*8+k, j);
                if(temp != back) m |= 1;
                if(temp ==back)
                putpixel(i*8+k, j, 2);
            }
 /*         fprintf(stdprn,"%c", m); */
            put_out(m);
            }
            fprintf(stdprn,"%s"  graph_fin),
        }
    }
    break;
```

```
    case HORIZONTAL:
    {
        xprint = 1000.0;
        yprint = 1000.0;
        strcpy(resolucion,"75");
        prstep = 9.6 * AspR;
        fprintf(stdprn,"%s%sR", graph_inic, resolucion);
        for(j=0; j<MaxX; j++)
        {
            fprintf(stdprn,"\x1B&a%-*.1fh%-*.1fV",
                    format(xprint), xprint,
                    format(yprint), yprint);
            yprint += prstep;
            fprintf(stdprn,"\x1B*r1A\x1B*b%dW", (int)(MaxY+4)/8);
            for(i=0; i<=MaxY/8; i++)
            {
                m = 0;
                for(k=0; k<8; k++)
                {
                    m<<=1;
                    if(getpixel(MaxX -j, i*8+k)) m++;
                }
                fprintf(stdprn, "%c", m);
            }
            fprintf(stdprn,"%s", graph_fin);
        }
        break;
    }
    }/*fin de switch */
    fprintf(stdprn,"\x0C\x1B&l0\x1B(8U\x1B(sp10h12vsb3T\x1B&l1H");
} /*fin de LJ_Graphic */

int format(double posicion)
{
    int ancho = 6;

    if(posicion < 1000.0) ancho--;
    if(posicion < 100.0)  ancho--;
    if(posicion < 10.0)   ancho--;
    return(ancho);
}
```

ANEXO B

En este anexo se presentan los datos iniciales para el algoritmo generador de mallas con elementos cuadrangulares del inciso 5.1, asociados con la figura 5.1.

Subregiones:

subregión	nodo 1	nodo 2	nodo 3	nodo 4	subregión	nodo 1	nodo 2	nodo 3	nodo 4
1	1	2	3	4	27	39	40	27	34
2	5	6	3	2	28	35	38	39	34
3	5	6	7	8	29	25	42	43	44
4	9	10	7	8	30	65	66	43	42
5	9	10	11	12	31	41	64	65	42
6	11	12	13	14	32	41	40	63	64
7	15	14	13	16	33	39	62	63	40
8	15	16	1	4	34	39	38	61	62
9	16	2	1	16	35	37	60	61	38
10	16	20	5	2	36	36	36	37	38
11	21	6	6	20	37	43	46	45	44
12	21	22	9	6	38	43	66	67	46
13	9	22	23	12	39	37	58	59	60
14	13	12	23	24	40	37	36	57	58
15	17	16	13	24	41	45	48	47	48
16	1	16	17	18	42	67	68	47	46
17	29	28	21	20	43	59	56	55	72
18	21	28	27	22	44	57	56	55	58
19	27	26	23	22	45	47	50	49	48
20	23	26	25	24	46	69	50	47	68
21	29	30	33	28	47	71	72	55	54
22	33	34	27	28	48	55	56	53	54
23	31	32	33	30	49	69	70	51	50
24	33	32	35	34	50	51	52	49	50
25	25	26	41	42	51	71	54	51	70
26	27	40	41	26	52	51	52	53	54

Coordenadas:

nodo	x	y	nodo	x	y
1	1.1	1.6	37	15.0	14.1
2	2.7	1.6	38	14.7	10.6
3	2.7	2.8	39	11.9	10.0
4	2.2	3.0	40	8.7	9.8
5	4.3	1.6	41	5.6	10.0
6	3.1	3.0	42	2.8	10.6
7	3.3	3.4	43	2.4	14.1
8	4.3	3.2	44	0.0	13.1
9	4.3	4.6	45	0.0	18.5
10	3.1	3.9	46	2.4	17.7
11	2.7	4.1	47	3.7	20.0
12	2.7	5.5	48	2.1	21.5
13	1.1	4.7	49	5.0	23.6
14	2.2	3.8	50	5.9	21.6
15	2.0	3.4	51	8.7	22.3
16	1.1	3.2	52	8.7	24.5
17	0.0	3.0	53	12.4	23.6
18	0.0	0.0	54	11.5	21.6
19	2.7	0.0	55	13.7	20.0
20	5.3	0.0	56	16.3	21.5
21	5.3	3.0	57	17.4	18.5
22	5.3	5.4	58	15.0	17.7
23	3.6	6.6	59	12.6	16.9
24	0.0	5.2	60	12.6	15.1
25	0.0	7.7	61	11.9	13.5
26	4.4	7.7	62	10.5	12.5
27	6.7	7.7	63	8.7	12.1
28	6.7	3.9	64	6.8	12.5
29	8.7	0.0	65	6.5	13.5
30	13.1	0.0	66	4.8	15.2
31	17.4	0.0	67	5.6	18.6
32	17.4	3.9	68	5.6	18.6
33	13.1	3.9	69	6.7	18.6
34	13.1	7.7	70	8.7	20.1
35	17.4	7.7	71	10.6	18.6
36	17.4	13.1	72	11.8	18.6

BIBLIOGRAFIA

Abhary K., 'An automatic mesh generation scheme for finite element models of box like structures.',*Computers and Structures*, Vol.31, No.4, pp.637-641, (1989).

Baehmann P.L. et al., 'Robust , geometrically based, automatic two dimensional mesh generation.', *Int. J. Numer. Methods. Eng.*, Vol. 24, pp. 1043-1078, (1987).

Barkakati N., '*The waite group's essential guide to Microsoft C.*', Howard W. Sams and Company, Indianapolis, Indiana (1989).

Burden R. L., Douglas J. C., *Análisis Numérico*. Grupo Editorial Iberoamérica, México D.F. (1985).

Bykat A., 'Design of recursive, shape controlling mesh generator.', *Int. J. Numer. Methods. Eng.*, Vol. 19, pp. 1375-1390, (1983).

Cavendish J. C and Hall C. A., 'A new class of transitional blended finite element for the analysis of solid structures.', *Int. J. Numer. Methods. Eng.* Vol 20, pp. 241-253, (1984).

Cavendish J. C, Field D. A. and Frey W. H., 'Automatic mesh grenération: A finite element/computer aided geometric design interface.', *THE MATHEMATICS OF FINITE ELEMENTS AND APPLICATIONS V.*, pp. 83-96, (1985).

Cavendish J. C, Field D. A. and Frey W. H., 'An approach to automatic three dimensional finite element mesh generation.', *Int. J. Numer. Methods. Eng.* Vol 21, pp. 329-347, (1985).

Cook W.A., 'Body oriented (natural) co-ordinates for generating three-dimensional meshes.', *Int. J. Numer. Methods. Eng.*, Vol. 8, pp. 27-43, (1974).

Cook R.D., Markus D. S. and Plesha, *Concepts and Applications of finite Element Analysis*. John Wiley, New York (1989).

Cheng et al. 'A parallel mesh generation algorithm based on the vertex label assignment scheme', *Int. J. Numer. Methods. Eng.* Vol 28, pp. 1429-1448, (1989)

Deljouie-Rakhshandeh K., 'An approach to the generation of triangular grids possesing few obtuse triangles', *Int. J. Numer. Methods. Eng.* Vol 29, pp. 1229-1321, (1990).

Durocher L. and Gasper A. 'A versatile two dimensional mesh generator with automatic bandwith reduccion.' *Computers and*

Structures., Vol. 10, pp. 561-575, (1979).

Ezzell B., *Graphics Programming in Turbo C 2.0.*, Addison-Wesley Publishing Company, (1989).

Field D. A.,'Algorithms for determining invertible two and three dimensional quadratic isoparametric finite element transformations.', *Int. J. Numer. Methods. Eng.* Vol 19, pp. 789-802, (1983).

Frey W.H., 'Selective refinement: a new strategy for automatic node placement in graded triangular meshes.', *Int. J. Numer. Methods. Eng.*, Vol. 24, pp. 2183-2200, (1987).

Frey W.H. and Field D.A., 'Mesh relaxation: a new technique for improving triangulations.', *Int. J. Numer. Methods. Eng.*, Vol. 31, pp. 1121-1133, (1991).

Ghassemi F. 'Automatic mesh generation scheme for a two or three dimensional triangular curved surface.'*Computers and Structures*, Vol.15, No.6, pp. 613-626, (1982).

Gordon W.J. and Hall C.A., 'Constructions of curvilinear co-ordinate systems and applications to mesh generation.' *Int. J. Numer. Methods. Eng.*, Vol. 7, pp. 461-477, (1973).

Ha K. H. 'C language for finite element programing.', *Computers and Structures*, Vol.37, No.5, pp. 873- 880, (1990).

Haber R. et al., 'A general two-dimensional, graphical finite element preprocessor utilizing discrete transfinite mappings.' *Int. J. Numer. Methods. Eng.*, Vol. 17, pp. 1015-1044, (1981).

Ho-Le K., 'Finite element mesh generation methods: A review and classification.', *Computer Aided Design.*, Vol. 20, pp 27-38, (1988).

Imafuku I., Kodera Y., Sayawaki M. 'A generalized automatic mesh generation scheme for finite element method.' *International Journal for Numerical methods in Engineering*, Vol.15, pp. 713-731, (1980).

Johnston B.P., Sullivan J.M., Jr and Kwasnik A., 'Automatic conversion of triangular finite element meshes to quadrilateral elements.', *Int. J. Numer. Methods. Eng.*, Vol. 31, pp. 67-84, (1991).

Joe B. and Simpson R.B., 'Triangular meshes for regions of complicate shape.', *Int. J. Numer. Methods. Eng.*, Vol. 23, pp. 751-778, (1986).

Kanaan V., *Technical* C *programming.*, Prentice Hall, Englewood Cliffs, New Jersey, (1989).

Kela A., 'Hierarchical octree approximations for boundary

representation-based geometric models.', *Computer Aided Design.*, Vol. 21, number 6, July/augost, pp. 355-362, (1989).

Kleinstreuer C., 'A triangular finite element mesh generator for fluid dynamic systems of arbitrary geometry.' *Int. J. Numer. Methods. Eng.*, Vol. 15, pp. 1325-1334, (1980).

Lehmann C. H., *Algebra*. Editorial LIMUSA, México D.F. (1983).

Liu Y. and Chen Y., 'A two dimensional mesh generator for variable order triangular and rectangular elements.', *Computers and Structures*, Vol.29, No.6, pp. 1033-1053, (1988).

Lo S.H., 'A new mesh generation scheme for arbitrary planar domains', *Int. J. Numer. Methods. Eng.*, Vol. 21, pp. 1403-1426, (1985).

Lo S.H., 'Delaunay triangulation of non-convex planar domains', *Int. J. Numer. Methods. Eng.*, Vol. 28, pp. 2695-2707, (1989).

Lo S.H., 'Generating quadrilateral elements on plane and over curved surfaces.' *Computers and Structures*, Vol.31, No.3, pp. 421-426, (1989).

Lo S. H., 'Automatic mesh generation and adaptation by using contours.', *Int. J. Numer. Methods. Eng.*, Vol. 31, pp. 689-707, (1991).

Moecardini A.O., Lewis B.A. and Cross M., 'ACTHOM - Automatic generation of triangular and higher order meshes.', *Int. J. Numer. Methods. Eng.*, Vol. 19, pp. 1331-1353, (1983).

Nguyen V.Ph., 'Automatic mesh generation with tetrahedron elements.', *Int. J. Numer. Methods. Eng.*, Vol. 18, pp. 273-289, (1982).

Park S. and Washam C.J., 'Drag method as a finite element mesh generation scheme.', *Computers and Structures*, Vol.10, pp. 343-348, (1979).

Pissanetzky S., 'KUBIK: an automatic three-dimensional finite element mesh generator.' *Int. J. Numer. Methods. Eng.*, Vol. 17, pp. 255-269, (1981).

Perucchio R. et al., 'Interactive computer graphic preprocessing for three-dimensional finite element analysis.', *Int. J. Numer. Methods. Eng.*, Vol. 18, pp. 909-926, (1982).

Perucchio R., Saxena M. and Kela A., 'Automatic mesh generation from solid models based on recursive spatial decompositions.', *Int. J. Numer. Methods. Eng.*, Vol. 28, pp. 2469-2501, (1989).

Rank E. and Babuška I. 'An expert system for optimal mesh design in the *hp*-version of the finite element method.', *Int. J.*

Numer. Methods. Eng., Vol. 24, pp. 2087-2106, (1987).

Sadek E.A., 'A scheme for automatic generation of triangular finite elements.', *Int. J. Numer. Methods. Eng.*, Vol. 15, pp. 1813-1822, (1980).

Schildt H., *Programación en Turbo C.*, BORLAN OSBORNE/McGraw-Hill, Madrid, (1988).

Schroeder W.J and Shephard M.S, 'A combined octree/Delaunay method for fully automatic 3-D mesh generation.', *Int. J. Numer. Methods. Eng.*, Vol. 29, pp. 37-35, (1990).

Segerling L. J., '*Applied Finite Element Analisis.*', John Wiley Sons, Inc. (1984).

Shaw R. D. and Pitchen R. G., 'Modification to the Suhara-Fukuda method of network generation.' *Int. J. Numer. Methods. Eng.*, Vol. 12, pp. 93-99, (1978).

Shephard M.S, Gallagher R.H. and Abel J.F , 'The synthesis of near-optimum finite element meshes with interactive computer graphics.' *Int. J. Numer. Methods. Eng.*, Vol. 15, pp. 1021-1039, (1980).

Stevens R.T., '*Graphics Programming in C.*' M and T Publising, Inc. Redwood City, California (1988).

Talbert J.A. and Parkinson A.R., 'Development of an automatic, two-dimensional finite element mesh generator using quadrilateral elements and Bezier curve boundary definition.' *Int. J. Numer. Methods. Eng.*, Vol. 29, pp. 1551-1567, (1990).

Taniguchi T., 'An interactive automatic mesh generator for the microcomputer.' *Computers and Structures*, Vol.30, No.3, pp. 715-722, (1988).

Thacker W. C., 'A brief review of techniques for generating irregular computational grids.', *Int. J. Numer. Methods. Eng.*, Vol. 15, pp. 1335-1341, (1980).

Yerry M. A. and Shephard M. S., 'A modified quadtree approach to finite element mesh generation.', *IEEEJ.*, pp. 39-46, Jan/Feb. (1983).

Zienkiewicz O. C. and Phillips D. V., 'An automatic mesh generation scheme for plane and curved surfaces by isoparametric coordinates.' *Int. J. Numer. Methods. Eng.*, Vol. 3, pp. 519-528, (1971).