UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
ISSN 0103-2577

ASPECTOS TEÓRICOS E COMPUTACIONAIS DA GERAÇÃO DE MALHAS QUADTREE

FERNANDO PACANELLI MARTINS

Nº 94

NOTAS

Série Computação

São Carlos – SP
Dez./2007

ASPECTOS TEÓRICOS E COMPUTACIONAIS DA GERAÇÃO DE MALHAS QUADTREE

FERNANDO PACANELLI MARTINS

Número USP: 5825419, e-mail: fmartins@icmc.usp.br
Mestrando em Matemática Aplicada e Computacional
Mecânica dos Fluidos Computacional

ICMC - Instituto de Ciências Matemáticas e de Computação
USP - Universidade de São Paulo
Av. Trabalhador São Carlense, 400, CP 668, CEP 13251-900

São Carlos
Novembro/2007

ASPECTOS TEÓRICOS E COMPUTACIONAIS DA GERAÇÃO DE MALHAS QUADTREE

FERNANDO PACANELLI MARTINS

Número USP: 5825419, e-mail: fmartins@icmc.usp.br
Mestrando em Matemática Aplicada e Computacional
Mecânica dos Fluidos Computacional

ICMC - Instituto de Ciências Matemáticas e de Computação
USP - Universidade de São Paulo
Av. Trabalhador São Carlense, 400, CP 668, CEP 13251-900

São Carlos
Novembro/2007

Conteúdo

Resumo

Uma interessante maneira de gerar malhas não-estruturadas em um domínio 2D se dá pela recursiva decomposição geométrica do mesmo, até que um determinado parâmetro de refinamento seja satisfeito em cada uma das partes decompostas, sendo tal estratégia o cerne da técnica quadtree.

Além dos conceitos e da metodologia básica inerente a esta ferramenta, discutiremos alguns mecanismos adicionais que podem ser utilizados para garantir um melhor controle de refinamento à malha final construída.

As idéias apresentadas são implementadas em linguagem C, originando o *QuadMesh2D*, que se constitui em uma flexível ferramenta para a geração de malhas quadtree.

Por fim, uma vasta quantidade de resultados numéricos são apresentados, os quais vêm em auxílio não só para ilustrar a técnica em questão, mas também, para confirmar a funcionalidade e alcance do gerador de malhas confeccionado.

Agradecimentos

O autor agradece à FAPESP pelo apoio financeiro de suas pesquisas no mestrado através do processo 06/04336-0.

Também aproveito a oportunidade para agradecer o professor Fabrício Simeoni de Sousa pela idéia de realização deste trabalho na disciplina "Tópicos em Análise Numérica I - Geração de Malhas", por ele ministrada no segundo semestre de 2007 no Instituto de Ciências Matemáticas e Computação da Universidade de São Paulo.

Capítulo 1

Introdução

1.1 Modelagem de figuras planas

A escolha de determinado gerador de malhas depende não só das propriedades desejadas para a malha final, mas, em primeira instância, da região sobre a qual ele atuará, isto é, do conjunto que queremos discretizar.

Obviamente, para gerar uma malha em uma região qualquer se torna imprescindível definir a mesma de forma não ambígua. Supondo que a porção a ser discretizada esteja contida em um conjunto universo, definir uma região sem ambiguidades consiste em encontrar uma relação que, para cada ponto do conjunto universo, seja capaz de informar se o mesmo é interno, externo ou pertence à fronteira da porção considerada. Em outras palavras, é necessário saber quais pontos pertencem ou não à região sobre a qual queremos gerar uma malha.

O gerador que abordaremos neste texto trabalha apenas com regiões 2D, de forma que podemos restringir o escopo de nosso raciocínio a porções bidimensionais[1]. Assim, levando este fato em consideração, faz-se interessante, inicialmente, definir os termos e conceitos relacionados aos tipos de regiões com as quais ele é capaz de lidar, sendo exatamente este o objetivo desta seção.

A partir de agora, todos os entes geométricos aqui considerados estarão situados em um único plano. Assim, chamamos de *vértice* a um ponto no plano considerado. Já uma *aresta* é um segmento de curva planar orientada, de comprimento não nulo, que não se auto-intercepta, partindo de um vértice (origem) e chegando a um outro (extremidade) distinto do primeiro. Será conveniente, para a consistência das definições, considerar um

[1] Conforme ficará subentendido ao longo deste texto, generalizações e restrições da técnica de geração de malhas quadtree para espaços, respectivamente, de dimensão maiores que dois e para espaços de dimensão um são perfeitamente possíveis. Para o caso 1D, a técnica bitree seria extremamente simples, consistindo na divisão recursiva de um intervalo universo em dois sub-intervalos, ficando implícita a utilização de uma árvore binária como estrutura de dados principal para a malha. Já no caso 3D, origina-se a técnica octree, que proporciona a divisão recursiva de um cubo universo em oito octantes. Embora o cerne da técnica não mude conforme a dimensão do domínio considerado aumenta, o custo computacional e a dificuldade de implementação crescem consideravelmente.

vértice como sendo uma *aresta degenerada*, a qual possui comprimento nulo e, por isso, não é uma aresta propriamente dita.

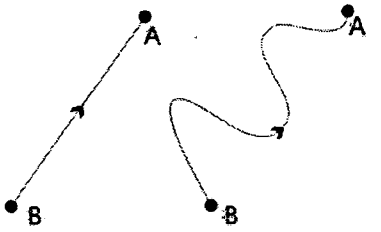


Figura 1.1.1: Exemplo de arestas com origem no vértice B e extremidade no vértice A.

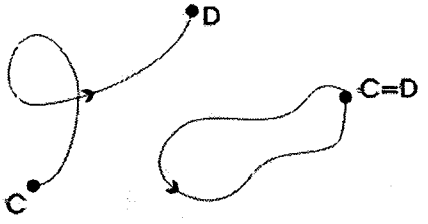


Figura 1.1.2: Exempo de curvas planares que não são arestas: enquanto a primeira se auto intercepta, a segunda tem origem e extremidade coincidentes.

A *fronteira de uma aresta* qualquer é composta pelos dois vértices que compõem sua origem e extremidade. Por outro lado, chamaremos de *ciclo* a uma sequência com mais de uma aresta, todas duas a duas coplanares, tais que apenas as consecutivas se interceptam, com isso ocorrendo apenas nos extremos, onde a extremidade de uma é a origem da seguinte.

Neste trabalho, por razões práticas, consideraremos que uma aresta é determinada por um segmento de reta. Desse modo, um ciclo passa a ser determinado por um *polígono* orientado, em que cada um de seus lados é formado por uma aresta orientada. Assim, chamaremos de *poligonal* a uma sequência com uma ou mais arestas, todas duas a duas coplanares, onde apenas as consecutivas se interceptam, com isso ocorrendo apenas nos extremos (a extremidade de uma deve ser a origem da seguinte), tal que a origem da primeira aresta e a extremidade da última aresta da sequência não devem coincidir.

A orientação em um ciclo nos permite distinguir duas regiões, uma interior e outra exterior ao mesmo. Assumiremos aqui, sem perda alguma de generalidade, que a distinção entre essas duas regiões se dá através da regra da mão direita. Nesta, ao dispor o polegar orientado na direção normal ao plano considerado e os dedos restantes apontando na

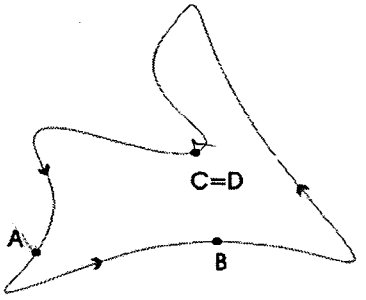


Figura 1.1.3: Exemplo de ciclo composto pelas arestas, duas a duas coplanares, CA, AB e BD, onde $C = D$.

mesma direção que a orientação de cada aresta, temos que o interior do ciclo é tomado pela região "abraçada" pela palma da mão direita. Obviamente, a região que fica do outro lado da fronteira passa a ser a região externa do ciclo.

O contorno definido pelo ciclo constitui uma linha a parte, que não pertence a nenhuma das duas regiões que delimita. O ciclo será a *fronteira externa* de seu conjunto de pontos interiores, e, de forma análoga, será a *fronteira interna* de seu conjunto de pontos exteriores.

Dada uma região delimitada por um ciclo externo (*fronteira externa da face*) a uma coleção de ciclos internos que não se interceptam (*fronteiras internas da face*), todos coerentemente orientados, chamamos de *face* ao conjunto de pontos que são interiores a todos os ciclos dados.

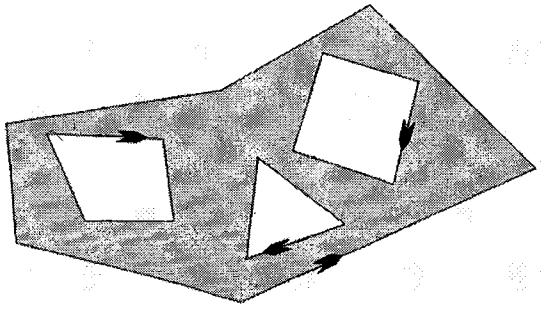


Figura 1.1.4: A região representada em amarelo constitui a face determinada e delimitada pelos ciclos considerados. Note que, para haver coerência na definição da face mostrada, faz-se necessário que o ciclo externo esteja orientado no sentido anti-horário, ao passo que os ciclos internos devem estar orientados no sentido horário. Tal convenção será fortemente usada neste trabalho, sendo ela um princípio básico para o correto funcionamento do gerador de malhas confeccionado.

Passamos, então, a denominar *domínio* uma região conexa por caminhos[2] do plano, composta por uma face (*pontos interiores do domínio*) e pela coleção de ciclos que definem a face (*fronteira do domínio*). O exterior de cada ciclo interno ao domínio é chamado de *buraco*, sendo uma porção do espaço que não pertence ao domínio, mas, está totalmente envolvido pelo mesmo.

Na Figura 1.1.4, o domínio é composto pela região em amarelo (face) juntamente com o ciclo externo (*fronteira externa do domínio*) e o grupo de ciclos internos (*fronteira interna do domínio*). Ainda nesta gravura, podemos observar a existência de três buracos, cada um associado a um dos três ciclos internos representados.

Tendo em vista os conceitos até então estabelecidos, os quais nos permitem tratar regiões planares que podem ser decompostas em vértices, arestas e faces, restringimos o universo das figuras planas àquelas que poderão ser discretizadas pela técnica quadtree (e, consequentemente, ser tratadas pelo *QuadMesh2D*).

Assim, uma vez já acertada a terminologia a ser empregada neste trabalho, estaremos, a partir de agora, interessados na geração da malha quadtree em um domínio. Neste ponto, salientamos que a exigência de conexidade por caminhos, imposta na definição do mesmo, é uma restrição do método em questão. Eventualmente, regiões desconexas poderão ser tratadas individualmente pelo método a ser considerado, bastando gerar uma malha, separadamente, em cada uma das componentes conexas que constituem a região desconexa.

Portanto, ao dizer que iremos gerar malhas sobre um domínio plano, queremos deixar claro que este representa uma figura plana que é univocamente descrita pela suas fronteiras externa (através de um ciclo externo orientado no sentido anti-horário) e interna (via uma coleção de ciclos disjuntos orientados no sentido horário e todos interiores ao ciclo externo inicialmente considerado). É exatamente dessa descrição do domínio que o algoritmo para geração da malha quadtree irá depender.

1.2 Contexto e motivação

Uma malha qualquer gerada em um domínio arbitrário pode ser enquadrada, invariavelmente, em dois grupos.

O primeiro deles diz respeito às *malhas estruturadas*, que são aquelas para as quais existe uma nítida correspondência biunívoca entre seus elementos com as entradas de uma matriz, com essa associação se dando tanto no nível lógico quanto espacial[3]. Uma malha desta categoria, embora facilite a aplicação de diversos procedimentos numéricos (graças ao acesso indexado direto a cada elemento da malha) e exija um custo um custo computacional "mínimo"(devido à correspondência biunívoca que pode ser traçada entre

[2]Dizemos que uma região do plano é conexa por caminhos se, dados dois pontos quaisquer a ela pertencentes, existe pelo menos uma curva, inteiramente contida na região, que liga tais pontos.

[3]Por correspondência no nível lógico queremos dizer que cada elemento da malha está associado biunivocamente a uma entrada da matriz. Já por correspondência no nível espacial deve ser compreendido que a mesma relação de adjacência entre os elementos da malha no domínio discretizado também ocorre com as entradas correspondentes (via a relação biunívoca considerada) da matriz.

elementos da malha e entradas de uma matriz), apresenta sérios inconvenientes no que diz respeito às possibilidades de refinamento e adaptação a geometrias mais complexas.

Já o segundo grupo de malhas faz menção às *não-estruturadas*, sendo aquelas que não se enquadram na classificação anterior justamente por não poderem ser obtidas pela contínua deformação de um *grid* regular. Estas, embora exijam uma estrutura de dados topológica (capaz de fornecer as relações de adjacências entre os elementos da malha, o que acarreta um custo extra no armazenamento computacional) e compliquem um pouco a aplicação de muitos procedimentos numéricos, mostram-se extremamente flexíveis na decomposição de geometrias irregulares e na aplicação de mecanismos que visam um refinamento concentrado em porções localizadas do domínio.

Neste contexto, usando a definição acima estabelecida, adiantamos que a técnica quadtree para geração de malhas se enquadra na categoria de malhas não-estruturadas. Apesar dessa categorização, podemos dizer que a malha quadtree apresenta uma característica, digamos, "híbrida", que lembra "resquícios evolutivos" de uma malha estruturada.

Isso porque se os elementos das malhas estruturadas gozam do fato de poderem ser colocados em correspondência biunívoca com as entradas de uma matriz, que é uma estrutura de dados extremamente prática, algo apenas ligeiramente menos prático ocorre nas malhas quadtree...

Apesar dos elementos de uma malha quadtree, de modo geral, não poderem ser colocados em correspondência biunívoca óbvia com as entradas de uma matriz, eles sempre podem ser colocados em correspondência biunívoca clara com parte das folhas de uma árvore quaternária[4].

Em tempo, uma árvore quaternária (quadtree) é um conjunto finito de elementos que pode ser definido de forma recursiva como sendo ou um conjunto vazio ou composto por um nó (elemento do conjunto considerado) ao qual estão associadas quatro outras árvores quaternárias, todas, simultaneamente, vazias ou não. Além disso, folha é um nó ao qual as quatro árvores quaternárias associadas são vazias. A Figura 1.2.1 nos dá uma visão pictórica de uma árvore quaternária, ilustrando o conceito recursivo enunciado.

De modo geral, note que a estrutura de árvore, embora não chegue a fornecer um acesso indexado direto a cada um de seus elementos (como ocorre no caso de uma matriz), possibilita uma busca eficiente a dados em seus nós. Portanto, concluímos esta seção afirmando que mesmo a malha quadtree sendo não-estruturada, ela está associada a uma estrutura de dados extremamente "bem comportada", o que ajuda na formulação dos algoritmos e no processo de refinamento[5].

---

[4]Conforme veremos, a árvore toda, e não apenas as folhas da árvore quaternária, são empregadas no processo de geração da malha quadtree. Já na etapa de utilização da malha construída, apenas as informações armazenadas em parte das folhas serão importantes (a saber, aquelas associadas a quadrantes de fronteira e a quadrantes internos). Assim, uma vez criada a malha, considerando essa relação biunívoca entre elementos da malha e folhas da árvore, se houver uma sequência fixa, definida especificamente pela aplicação considerada, através da qual os elementos da malha serão acessados repetidas vezes nessa mesma sequência, faz-se possível "jogar"a árvore fora e armazenar as folhas convenientes em uma lista, o que proporcionaria quase que um acesso indexado direto aos elementos da malha quadtree.

[5]Esta ajuda se associa à facilidade para se caminhar nesta estrutura de dados a partir de procedimen-

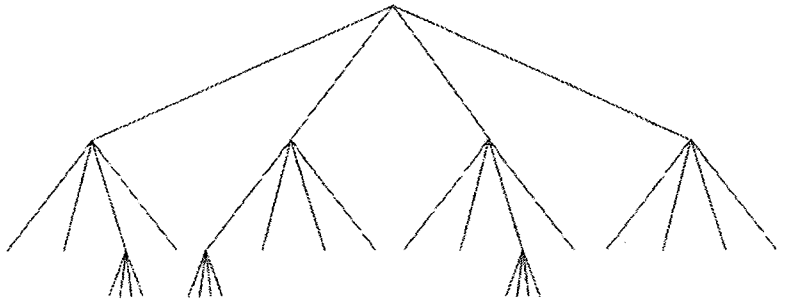


Figura 1.2.1: Neste exemplo de árvore quaternária, perceba que todo nó é "pai" ou de zero ou de quatro nós "filhos", satisfazendo a definição recursiva dada.

tos recursivos, os quais, para esta aplicação específica, contrariam o senso comum e são relativamente eficientes.

Capítulo 2

A técnica quadtree para geração de malhas

2.1 Introdução

Consideremos, inicialmente, um domínio qualquer onde desejamos gerar uma malha tipo quadtree. Para isso, faz-se necessário que tenhamos a coordenada de cada vértice que compõe o ciclo externo e os eventuais ciclos internos do mesmo, bem como o número de vértices presentes em cada ciclo.

Nestes termos, podemos criar, basicamente, dois tipos de listas: a primeira sendo composta com a coordenada dos vértices que definem o ciclo externo e, a segunda, indexando cada um dos ciclos internos definidos. A cada posição da lista de ciclos internos associamos uma nova lista, da mesma natureza que aquela associada ao ciclo externo, mas, agora, armazenando em cada posição a coordenada de cada um dos vértices que compõem o respectivo ciclo interno. Por razões de simplicidade e a fim de fazermos ecoar na prática o que a definição de ciclo diz, faz-se conveniente que cada lista de coordenadas associada a um ciclo comece e termine com o mesmo vértice, de modo que o ciclo seja, de fato, fechado.

Conforme já foi dito, é imprescindível que a lista que define o ciclo externo tenha seus vértices dispostos em sentido anti-horário, enquanto que, para cada ciclo interno, a lista de seus vértices é construída considerando a enumeração dos mesmos no sentido horário. Desse modo, o domínio sobre o qual desejamos criar a malha ficará consistentemente definido.

Portanto, uma vez que já padronizamos como deverá ser a entrada do domínio para o gerador de malhas quadtree, podemos passar à discussão do método em si.

Esta técnica para geração de malhas é baseada em um procedimento de representação de figuras 2D por recursiva decomposição do domínio em quadrantes, de modo que o objeto passa a ser representado por um conjunto de células quadradas, que são os quadrantes propriamente ditos. Conforme mostraremos agora, cada quadrante pode ser associado, de forma biunívoca, a certo nó na árvore quaternária.

Inicialmente, o domínio em que a malha será gerada é inscrito em um quadrado

fechado, o qual passamos a chamar de *universo do modelo* ou *quadrante universo*. Sendo este o primeiro e maior quadrante, associamos o mesmo à raiz de uma árvore quaternária.

No que segue, o quadrante universo é dividido em quatro novos quadrantes, cada um com lado igual a metade do lado do quadrante original. A partir disso, o método quadtree de representação do domínio passa a ser dado de forma recursiva, sendo decidido, para cada um dos quatro quadrantes recém criados, se ele será ou não subdividido em quatro novos quadrantes com lado igual à metade do comprimento do quadrante dividido, com esta decisão voltando a ser aplicada, recursivamente, a cada um dos quadrantes obtidos em uma subdivisão.

Por questões óbvias, apenas quadrantes que interceptam o domínio precisam voltar a ser subdivididos, já que estamos interessados justamente na representação do domínio, e não de porções externas ao mesmo (ainda que internas ao quadrante universo).

Assim, um quadrante qualquer pode se classificado como *exterior* ou *vazio* (se não intercepta nenhuma porção do domínio), *interior* (quando intercepta o domínio), ou ainda, de *fronteira*, caso intercepte uma fronteira do domínio. Um quadrante que volta a ser subdividido é chamado de *quadrante de continuação*, ao passo que um quadrante que não mais é subdividido é dito ser um *quadrante terminal*.

Considerando que um quadrante representa um nó na árvore, subdividi-lo em quatro novos quadrantes corresponde à inserção de quatro filhos ao respectivo nó na árvore quaternária, cada um associado a um quadrante, conforme ilustra a Figura 2.1.1. Por outro lado, um quadrante que não mais é subdividido, isto é, um quadrante terminal, passa a corresponder a uma folha na árvore quaternária.

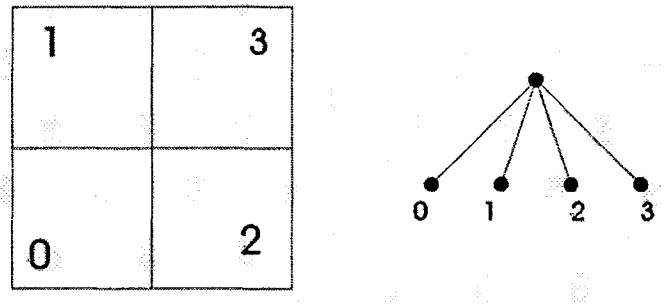


Figura 2.1.1: Correspondência entre os quatro filhos de um nó pai na árvore quaternária e a subdivisão de um quadrante pai em quatro novos quadrantes filhos: a cada quadrante é associado um nó, e a cada nó fazemos corresponder um quadrante, conforme a nomenclatura dada. Aliás, essa nomenclatura de numeração de quadrantes ou nós irmãos é usada durante todo este trabalho, inclusive na implementação do *QuadMesh2D*.

O processo recursivo de criação de quadrantes deve ser orientado pelo nível de representação do domínio, através da árvore quaternária e da estrutura de quadrantes, que desejamos alcançar. Obviamente, quanto mais subdivisões e, consequentemente, mais

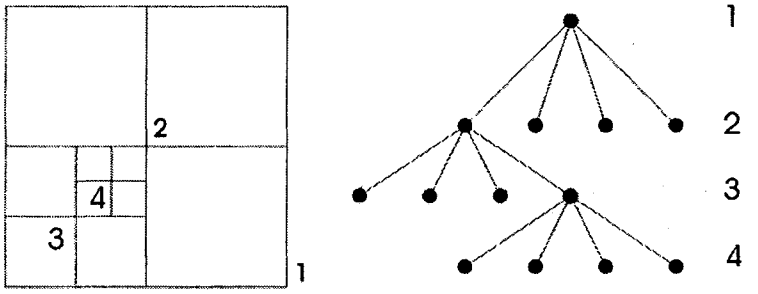


Figura 2.1.2: Ilustração de um processo de subdivisão do domínio em quadrantes e respectiva árvore quaternária criada. Perceba a correspondência existente entre a quantidade de subdivisões associada a um quadrante e o nível atingido pelo respectivo nó na árvore quaternária. A associação de cada porção do domínio a um nível mínimo a ser alcançado na árvore é que determinará a parada do processo recursivo de criação da quadtree ou subdivisão dos quadrantes.

níveis na árvore, melhor seria a representação do objeto (em uma situação limite, com todos os quadrantes sendo infinitamente divididos, cada folha da árvore passaria a corresponder a um único ponto do quadrante universo).

No entanto, é evidente que, em termos práticos, em algum momento devemos parar de subdividir os quadrantes, pois, tal processo não pode ser prorrogado indefinidamente. Em outras palavras, a cada porção do domínio devemos estabelecer níveis mínimos que um quadrante terminal que contenha essa região deve apresentar, de modo que, uma vez alcançado esse nível mínimo, tal quadrante para de ser subdividido. Em outras palavras, dado um quadrante, ele voltará a ser subdivido enquanto o nível mínimo de representatividade exigido por algum "ente"nele contido for maior que o nível do nó associado na quadtree.

Na verdade, o que fazemos para especificar tais níveis de subdivisão de um quadrante é associar níveis mínimos a cada um dos vértices e arestas que estão na fronteira do domínio considerado. Assim, associando um nível mínimo, digamos, $p_v \in \mathbb{N}$ a cada vértice (podendo variar para cada vértice), exigimos que qualquer quadrante terminal que intercepte tal vértice só poderá estar associado a uma folha na árvore quaternária se o nível dessa folha for, no mínimo, p_v. De modo análogo, podemos associar um nível mínimo a cada aresta, digamos, $p_a \in \mathbb{N}$, exigindo que um quadrante terminal que intercepta alguma porção dessa aresta só poderá estar associado a uma folha na árvore quaternária que apresenta nível, no mínimo, p_a.

Em nosso modelo de representação das fronteiras através de listas de vértices, associamos o valor p_v diretamente ao vértice armazenado, ao passo que, como não guardamos explicitamente as arestas (com estas implicitamente determinadas por cada par de vértices consecutivos da lista), o parâmetro p_a deve ser associado ao vértice que é a origem da aresta à qual tal parâmetro se refere.

Com os parâmetros p_v e p_a especificados para cada vértice e aresta da fronteira, estabelecemos os níveis mínimos de subdivisão associados às fronteiras do domínio. Entretanto, em vários contextos, torna-se necessário estabelecer níveis mínimos de refinamento também em regiões interiores ao domínio, ou, até mesmo níveis diferenciados de refinamento ao longo de uma mesma aresta da fronteira. Como não guardamos em nossa representação nenhuma informação explícita a essas regiões, faz-se conveniente a utilização de *poligonais de controle*.

Cada poligonal de controle (podendo existir tantas quanto forem necessárias) deverá ser representada por uma lista de vértices, sendo que, para cada um deles, devemos armazenar sua coordenada e os parâmetros p_v e p_a, de forma idêntica ao que fizemos para as listas que definem os ciclos. Uma poligonal de controle, no entanto, não precisa definir um ciclo, podendo, inclusive, ser constituída por um único ponto de controle (neste caso, a informação p_a associada a esse vértice deixa de ter sentido.

Além disso, não podemos nos esquecer que as listas associadas às poligonais de controle servem apenas para imprimir um nível de refinamento arbitrário a regiões específicas (onde os parâmetros de refinamento associados às fronteiras não são suficientes para prorrogar convenientemente o processo recursivo de subdivisão dos quadrantes), e não para delimitar uma fronteira do domínio[1].

Portanto, uma quadtree é utilizada para representar um objeto através de quadrantes terminais (os quais são criados por recursiva subdivisão de um quadrante universo que engloba o domínio todo). Cada um desses quadrantes terminais pode ser enxergado, em última instância, como uma célula de uma possível malha...

Embora os quadrantes da quadtree, *a priori*, não sejam conformados às fronteiras que definem o domínio, eles dividem o domínio em elementos que são finitos, sendo exatamente isto o que se espera, em um primeiro momento, de um gerador de malhas. Logo, a técnica de geração de malhas quadtree é baseada nos elementos finitos criados com uma representação de um domínio via uma árvore quaternária. Obviamente, para a confecção da malha em si, todo quadrante de fronteira deverá ser "recortado" de modo a se ajustar às fronteiras do domínio (evitando o famoso efeito escada).

Já no que diz respeito à árvore quaternária em si, tendo em vista o contexto desta aplicação, existe um conjunto básico de informações que cada nó deverá armazenar. Como não poderia ser diferente, uma delas diz respeito às coordenadas dos extremos do quadrante ao qual o nó está associado. Além disso, faz-se conveniente reservar dois campos que armazenarão, respectivamente, o nível atingido pelo nó na árvore como um todo e a posição do quadrante associado ao nó com relação ao domínio. Para que se possa "passear" pela árvore tanto na direção descendente (da raiz para as folhas - percurso *top-down*) quanto

[1]Note que aumentar o nível mínimo de subdivisão exigido por uma aresta (seja ela de fronteira ou de controle), apenas contribui para a criação de novos quadrantes cada vez mais próximos da própria aresta, em nada influindo no refinamento em porções do domínio remotas à mesma. Conforme veremos, o processo de balanceamento local da quadtree irá ocasionar novas subdivisões (as quais objetivam apenas equilibrar, segundo uma tolerância estabelecida, o desnível existente entre quadrantes vizinhos), o que embora ajude no processo de transição suave do refinamento, não dispensa o uso de poligonais de controle como meio de garantir localidades com maior nível de refinamento no domínio.

ascendente (das folhas para a raiz - percurso *bottom-up*), faz-se imprescindível armazenar, respectivamente, quem são os filhos de um dado nó, bem como quem é seu pai. Além disso, para os quadrantes terminais e de contorno, já visando seu "recorte" para conformação à fronteira do domínio no processo de criação da malha, faz-se imprescindível armazenar uma lista com as coordenadas dos vértices que definem o "ciclo" determinado pelo elemento de fronteira da malha.

Aconselhamos que o leitor, neste momento, antes de prosseguir com a leitura, examine os trechos do código fonte do *QuadMesh2D* relativos aos detalhes já abordados. Esses detalhes dizem respeito à definição das estruturas de dados empregadas (para representar o domínio, as poligonais de controle e a árvore quaternária) e aos procedimentos que manipulam tais estruturas em estado básico (criação das listas de vértices, ciclos internos e poligonais de controle, bem como a criação do quadrante universo - raiz da árvore - e alocação de novos nós para a árvore quaternária - onde está implícita a subdivisão do quadrante pai em quatro quadrantes filhos).

Por fim, encerramos esta introdução à técnica quadtree decompondo esta metodologia para geração de malhas em dois momentos consecutivos:

1. construção da árvore quaternária;

2. construção da malha quadtree.

Chamamos a atenção do leitor para a diferença existente entre a quadtree e a malha quadtree. A primeira busca a representação do domínio considerado, sendo, basicamente, a tarefa que deve ser executada na fase (1) acima mencionada. Já a malha quadtree consiste em aproveitar o modo de enxergar o domínio do ponto de vista de uma quadtree bem construída em (1) para criar uma coleção de elementos finitos, conformados às fronteiras, os quais discretizarão o domínio, sendo este o papel da fase (2) citada.

No que segue, abordaremos com detalhes cada uma dessas etapas, salientando a importância e a necessidade de cada procedimento para a técnica quadtree como um todo.

2.2 Construção da árvore quaternária

A geração da malha quadtree propriamente dita é feita tomando como entrada uma árvore quaternária conveniente. Tal malha é obtida a partir do momento em que temos a garantia que o domínio não só é minimamente representável pela árvore quaternária estabelecida (*quadtree mínima*), mas também, que tal quadtree induzirá a uma malha de elementos finitos com desníveis de refinamento[2] menores que uma tolerância prescrita (*quadtree balanceada*).

A necessidade de conseguirmos uma quadtree mínima é bastante intuitiva se pensarmos da seguinte forma: como a malha é gerada a partir da visão que a quadtree tem do domínio,

[2] A medida do desnível de refinamento entre quadrantes adjacentes da malha nada mais é do que uma medida de suavidade na transição de elementos de tamanhos distintos.

só podemos esperar uma malha minimamente boa se esse ponto de vista enxergar bem o domínio!

Em outras palavras, para os níveis mínimos de refinamento exigidos em cada porção do domínio, a árvore que resulta do menor número de subdivisões adicionais dos quadrantes, de modo a ser capaz de representar o domínio desejado de forma conveniente para a geração da malha, é a *quadtree mínima*. Assim, a quadtree mínima está intimamente relacionada ao domínio que é representado via quadtree e, também, aos níveis de refinamento exigidos.

Por exemplo, considere um domínio inserido no quadrante universo. Se pensarmos em um caso extremamente patológico no qual associamos nível mínimo de refinamento igual a um em cada porção do domínio, a quadtree que deveria ser construída estaria restrita a uma raiz, isto é, a um único quadrante. Por outro lado, é evidente que esta representação quadtree não é boa (qualquer que seja o domínio) para a geração da malha, pois, induziria a um único elemento finito (afinal, também só temos um quadrante)! Para este caso, bem como para outros menos patológicos, a quadtree mínima se encarrega de "completar" a representação quadtree exigida do domínio de forma a obtermos uma árvore quaternária apropriada para a geração da malha quadtree.

Formalmente, a construção da quadtree mínima se dá em três etapas, sendo que ao final de cada uma delas obtemos uma árvore com um número de nós maior ou igual que a árvore previamente considerada. Isto significa que as etapas de obtenção da quadtree mínima vão subdividindo o domínio gradualmente, e um quadrante, uma vez subdividido, nunca vem a ser "colado".

Assim, cada etapa de confecção da quadtree mínima resultará em uma quadtree intermediária, a qual garante uma dada propriedade. Sequencialmente, são elas:

Quadtree básica por vértices: a partir do quadrante universo, complementa o processo de divisão recursiva do domínio até que tenhamos apenas um vértice de fronteira em cada quadrante associado a uma folha; isto significa que um quadrante terminal só poderá abrigar na região por ele abrangida um único vértice do conjunto de listas que definem a fronteira do domínio; note, porém, que é possível a um dado vértice estar, simultaneamente, em até quatro quadrantes, bastando que esteja na fronteira comum dos mesmos.

Quadtree restante: esta é o resultado da ampliação da quadtree já existente, visando satisfazer, para cada porção do domínio, o nível mínimo de refinamento exigido; assim, cada quadrante terminal da quadtree existente deverá ser subdividido enquanto abranger algum "ente" que exija um nível mínimo de refinamento maior que o nível já alcançado pelo nó associado.

Quadtree básica por vértices: esta etapa deverá garantir que cada quadrante terminal da quadtree venha a ser interceptado por, no máximo, uma aresta de fronteira (seja esta fronteira interna ou externa); uma óbvia exceção deve ser feita aos quadrantes que abrigam extremo e origem de arestas de fronteira que são consecutivas, os quais poderão ser interceptados apenas por essas duas arestas; de forma parecida

ao que acontece com a quadtree básica por vértices, na árvore básica por arestas, um ponto de determinada aresta poderá estar em vários quadrantes terminais de forma simultânea, mas, dado um quadrante terminal, poderá existir apenas uma aresta que o intercepta.

Desse modo, uma vez construída essas três ampliações da quadtree (que, inicialmente, era composta apenas pelo quadrante universo), já temos uma representação do domínio minimamente plausível, capaz de representá-lo de forma conveniente para que pudéssemos passar, agora mesmo, para a etapa de construção da malha quadtree propriamente dita (uma vez que, até agora, apenas propiciamos o crescimento da árvore, sem fazer qualquer menção ou guardar qualquer informação relativa à malha na mesma).

No entanto, existe um outro aspecto que também deve ser levado em conta na construção da quadtree, já tendo em vista a malha de elementos finitos que será resultante, e que pode ocasionar a inserção de novos nós (novos quadrantes subdivididos) na mesma. Considerando a quadtree mínima já construída, como em sua confecção foi levada em conta apenas informações existentes no interior de cada quadrante, pode ocorrer que entre quadrantes vizinhos ocorra um desnível[3] bastante acentuado.

Neste momento, salientamos que a diferença de nível maior que zero entre, pelo menos, alguns quadrantes vizinhos, é inevitável pela técnica quadtree. Com efeito, se partirmos para a ignorância e exigirmos que *todas* as folhas da quadtree estarão em um nível $n \in \mathbb{N}$, obviamente que o desnível entre quaisquer dois quadrantes vizinhos será nulo, mas, por outro lado, estaremos criando uma árvore quaternária completa de nível n, a qual resultaria em uma malha uniforme[4] no quadrante universo tal que cada quadrante terminal viria a ser um quadrado de lado $2^{1-n}l$, com l sendo o lado do quadrante universo.

Assim, se não for para permitir desníveis, podemos abandonar a técnica quadtree e usar a simples geração de uma malha uniforme no domínio...

Obviamente que queremos insistir com a técnica quadtree, e assim, passaremos a ter que conviver com o desnível entre quadrantes. Apesar disso, segue que podemos limitar o máximo desnível existente entre quadrantes vizinhos em $t \in \mathbb{N}$, por meio do *balanceamento local da quadtree.*

Portanto, passamos a chamar de *quadtree balanceada* àquela resultante da quadtree mínima através de novas subdivisões (dos quadrantes terminais) capazes de tornar o desnível máximo entre quadrantes terminais vizinhos, no máximo, igual a um valor t. Note que tais subdivisões adicionais não devem aumentar o nível máximo que fora atingido pela quadtree mínima, mas, apenas, "nivelar" a quadtree já existente segundo a tolerância $t \in \mathbb{N}$ considerada.

Assim, a idéia por trás do balanceamento local é propagar, com um decaimento t a cada quadrante, o nível de refinamento atingido por um dado quadrante da quadtree mais refinado que seus quadrantes vizinhos. Desse modo, conforme já havíamos adiantado no

[3]Por desnível entre quadrantes vizinhos entendemos a diferença de nível existente entre os nós associados aos mesmos na árvore quaternária.

[4]Note que, com isso, estamos dizendo que uma malha uniforme é um caso particular da malha quadtree criada a partir de uma árvore quaternária completa.

início desta seção, a medida do desnível de refinamento entre quadrantes adjacentes da malha passa a ser usada como um meio de garantir uma suavidade mínima na transição de elementos, impedindo que quadrantes terminais associados a folhas de alto nível na quadtree façam fronteira com quadrantes terminais associados a folhas de baixo nível na quadtree.

Em outras palavras, a tolerância t garantirá que os elementos finitos da malha quadtree venham a ter uma proporção aceitável entre o comprimento de seus lados, ou seja, uma boa razão de aspecto.

Abordamos, nesta seção, a criação da árvore quaternária que induzirá a uma malha de elementos finitos através de seus quadrantes terminais, sendo o objetivo da próxima mostrar, exatamente, como isso deve ser feito. Antes, porém, voltamos a recomendar que o leitor interrompa, momentaneamente, a sequência do capítulo e verifique, no código fonte do *QuadMesh2D*, a implementação dos algoritmos responsáveis por executar as tarefas até então delineadas.

2.3 Construção da malha quadtree

Na árvore quaternária, resultante da construção proposta na seção anterior, ainda não temos nenhuma informação associada à malha quadtree propriamente dita. Assim, mostraremos agora uma maneira de aproveitar o conjunto de quadrantes terminais existentes a fim de criarmos uma decomposição do domínio em elementos finitos. Neste processo, os pressupostos tomados para a obtenção da quadtree mínima são indispensáveis.

Nestes termos, assumindo a quadtree então existente, o processo de confecção da malha quadtree a partir da árvore quaternária pode ser decomposto em três momentos.

Em primeira instância, faz-se necessário discretizar as fronteiras do domínio segundo suas interceptações com os quadrantes terminais da quadtree existente. Para isso, precisamos definir o conceito de *box de intersecção* de um dado ponto da fronteira do domínio como sendo o conjunto de quadrantes terminais da quadtree que interceptam o ponto considerado. A *fronteira do box de intersecção* é dada pelo subconjunto de lados dos quadrantes que compõem o box que não interceptam o ponto gerador do box.

Perceba que, pelo critério de construção da quadtree básica por vértices, no box de intersecção de um dado vértice da fronteira não poderá existir nenhum outro vértice da fronteira, pois, caso isso ocorresse, teríamos dois vértices de fronteira distintos em um mesmo quadrante e, assim, este não poderia ser um quadrante terminal. Da mesma forma, pela construção da quadtree básica por arestas, o box de intersecção de um ponto qualquer da fronteira só poderá ser interceptado pelas arestas que passam por este ponto, pois, caso isso não ocorresse, algum dos quadrantes que compõem o box não poderia ser terminal.

Para que o contorno do domínio possa ser discretizado, o que devemos fazer, então, é, para cada aresta de fronteira, percorrer tal aresta a partir de sua origem até que a primeira intersecção com a fronteira do box (determinado pela origem de tal aresta) seja encontrada. O quadrante do box ao qual este ponto de intersecção pertence passa, com

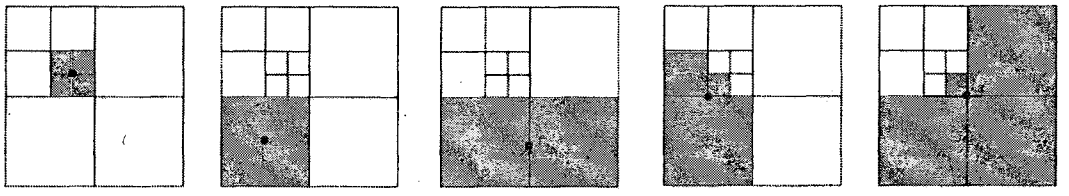


Figura 2.3.1: Exemplos de alguns possíveis boxes de intersecção de um dado ponto de fronteira. Um dado box poderá ser composto por, no máximo, quatro quadrantes terminais. Além disso, se a tolerância para o balanceamento local for tomada $t = 1$, apenas os três primeiros casos de boxes poderão ocorrer. Já se tomamos $t > 1$, um número maior de possibilidades poderá surgir, estando, entre elas, os dois últimos boxes da direita.

isso, a ser considerado um quadrante de fronteira. Neste momento, o ponto inicial (que originou o box) e o ponto encontrado deverão compor a primeira informação discreta desta aresta a ser armazenada.

No que segue, o último ponto de intersecção encontrado (o qual pertence à aresta considerada no início desta iteração do procedimento) passa a ser o "centro" de um novo box de intersecção, e a metodologia acima volta a ser repetida, tendo em vista alcançar a próxima intersecção da aresta corrente com o box "centrado" no último ponto de intersecção. Isto deverá ocorrer até que aresta inicialmente considerada "termine", isto é, enquanto a extremidade da aresta de fronteira inicialmente tomada não pertencer ao box de intersecção corrente. Quando isto ocorre, devemos armazenar este último subsegmento da aresta de fronteira (cujo extremo coincide com o da aresta em questão) na nossa base de dados, estabelecer que este quadrante é de fronteira e, para o extremo da aresta encontrado (que é a origem da aresta seguinte do ciclo), dar continuidade ao processo que fora executado para a primeira aresta, até todos os ciclos que compõem a fronteira do domínio sejam discretizados.

No processo de transição de uma aresta de fronteira para outra devemos tomar cuidados adicionais, no momento da implementação, a fim de não armazenar o vértice comum a tais arestas consecutivas mais de uma vez na base de dados adotada para representar a discretização do contorno, a menos que ele pertença a mais de um quadrante (isto é, ele deve ser armazenado mais de uma vez apenas no caso do subsegmento do qual ele é extremo e do subsegmento do qual ele é origem ficarem em quadrantes distintos).

Além disso, após a discretização de todas as arestas que compõem um dado ciclo, devemos tomar o cuidado adicional de verificar (dependendo da maneira que estivermos armazenando o contorno discreto e sua orientação) se a poligonal que discretiza a porção do contorno associada ao box de intersecção do primeiro vértice mantém a orientação da fronteira, com a qual deve coincidir. Isto porque, embora um ciclo não tenha início e nem fim, a discretização começou por uma "primeira" aresta, de forma que o ponto discretizado antes do "primeiro" vértice é, por tal procedimento, apenas o último a entrar na base de

dados, quando, na verdade, para manter a orientação do contorno, deveria ter sido o primeiro a ser incluído. Com isso, se esta base de dados estiver sendo armazenada local[5] e sequencialmente[6] a cada quadrante de contorno, para aquele que envolve o primeiro vértice, o ponto por onde deveria se iniciar a poligonal que intercepta tal quadrante, será, na verdade, o último a ser adicionado à sequencia orientada de vértices associada à poligonal que discretiza o contorno. Portanto, fica claro que devemos tomar cuidado com esse detalhe da implementação, a fim de garantirmos que a discretização do contorno de um ciclo não altere a orientação a ele inerente em porção alguma do mesmo.

Repare que, no processo acima descrito, discretizamos a fronteira do domínio e setamos quais são os quadrantes de fronteira. Assim, definimos uma poligonal no interior de cada quadrante de fronteira, dividindo o mesmo em duas porções, sendo uma interior e a outra exterior ao domínio, conforme ilustra a Figura 2.3.2 para algumas disposições possíveis.

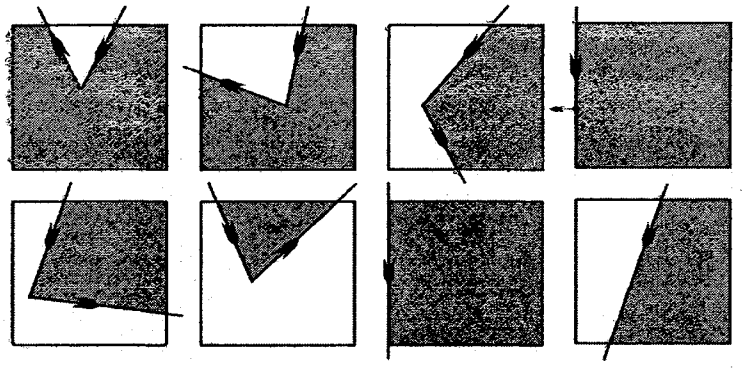


Figura 2.3.2: Configurações admissíveis para um elemento de contorno da malha quadtree, com a poligonal orientada sendo a discretização da fronteira do domínio que intercepta o quadrante de contorno. A porção verde de cada figura representa a parte do quadrante de fronteira que é interior ao domínio, revelando o fato que um elemento de contorno da malha quadtree poderá ser desde um triângulo até um heptágono. Note que nada impede a porção do quadrante de contorno exterior ao domínio ser vazia, com este caso ocorrendo quando a poligonal definida coincide com a fronteira do quadrante.

Assim, considerando a poligonal orientada que intercepta o quadrante de fronteira, faz-se necessário especificarmos quais são os vértices do quadrante de contorno que deverão ser adicionados, sequencialmente, a esta poligonal, a fim de que a mesma possa vir a ser fechada, e, com isso, definir o elemento finito de contorno da malha quadtree. Na

[5]Por local entendemos que armazenamos a poligonal que intercepta o quadrante de contorno, de alguma maneira, neste mesmo quadrante (isto é, na folha da árvore quaternária).

[6]Com o termo sequencial queremos dizer que a orientação da poligonal está associada à ordem pela qual tais vértices são inseridos na base de dados. Esta, no *QuadMesh2D*, nada mais é do que uma lista sequencial, alocada dinamicamente apenas para os nós da árvore quaternária que são, sabidamente, terminais e de fronteira.

realização deste serviço consiste o segundo momento de criação da malha quadtree a partir da árvore quaternária.

Um meio de realizar esta segunda etapa pode ser através do seguinte mecanismo: considerando sempre o último vértice da poligonal (que deve estar sobre a fronteira do quadrante), devemos verificar, inicialmente, se a poligonal que definirá o elemento finito de fronteira poderá vir a ser fechada, o que finalizaria tal processo para este elemento de contorno. Isto ocorre quando os extremos da poligonal estão sobre um mesmo lado do quadrante de contorno e o segmento com origem na extremidade da poligonal e extremidade na origem da mesma (o qual deve estar sobre a fronteira do domínio) possui a mesma orientação que a inerente à fronteira. Caso ainda não seja fechada, um novo vértice do quadrante de fronteira deverá ser inserido na poligonal.

O critério de inserção para este novo vértice será o seguinte, o qual é subdividido em duas situações: caso o último vértice da poligonal seja um vértice de "canto" do quadrante, isto é, caso pertença a dois lados do quadrante, testamos qual dos extremos destes lados está no interior do semi-plano interno (segundo a regra da mão direita) à reta determinada pelo último segmento da poligonal corrente; já no caso do último vértice da poligonal não estar sobre um "canto" do quadrante, temos que os possíveis candidatos para a inserção são os dois cantos deste mesmo lado, ganhando aquele que estiver na semi-reta orientada com origem aberta no penúltimo ponto da poligonal e que passa pelo último ponto corrente da mesma.

Pelo procedimento acima descrito, inserimos ponto a ponto os novos vértices da poligonal, só parando quando conseguirmos fecha-la, com tal poligonal fechada definindo um elemento finito de fronteira da malha quadtree. Obviamente, o mecanismo acima descrito deverá ser realizado para todos os quadrantes de contorno da quadtree.

Perceba, portanto, que, neste estágio de confecção da malha quadtree, já discretizamos os contornos e definimos os elementos de fronteira. Resta, agora, diferenciar, dentre os quadrantes terminais que ainda não foram classificados, quais são internos ao domínio (e deverão ser adicionados à malha) e quais são externos ao mesmo (e não poderão ser aproveitados), sendo este o terceiro e último momento de criação da malha quadtree.

Resolver tal problema é bastante simples se repararmos que um quadrante ainda não classificado com tipo fronteira deverá ter todos seus pontos interiores ou internos ou externos ao domínio. Isso porque um quadrante exterior pode ter sua fronteira interceptando a fronteira do domínio e, mesmo assim, não ter sido classificado como de fronteira, perdendo para seu adjacente que não só tinha parte de um lado sobre a fronteira do domínio como também todos os demais pontos interiores ao mesmo.

Logo, se um quadrante ainda não classificado como tipo fronteira tiver pelo menos um ponto interno externo ao domínio, podemos concluir que tal quadrante deverá ser externo à malha, e, caso contrário, deverá ser interno. Portanto, um meio existente para encontrar a posição relativa de um quadrante que não é de fronteira consiste em verificar se seu baricentro é interno ou externo ao domínio, implicando que o quadrante deverá receber a mesma classificação.

Desse modo, reduzimos o problema de determinar a posição relativa de uma região ao serviço de estabelecer a posição relativa de um dado ponto dessa mesma região. Para

isso, podemos, agora, recorrer ao tradicional algoritmo matemático que checa se um dado ponto é interno ou não a um domínio.

Neste algoritmo, devemos tomar um *ponto remoto*, que é um ponto externo ao ciclo externo que define o domínio. Feito isso, e considerando o baricentro do quadrante a ser testado (ponto de teste), determinamos um *segmento teste*. No que segue, para cada uma das arestas que estão nas fronteiras externa e interna ao domínio, verificamos se a mesma intercepta ou não o segmento teste. Para que uma intersecção não seja contada duas vezes, devemos supor que as arestas apresentam origem fechada e extremo aberto. Além disso, quando a aresta fica sobre o segmento teste, tal coincidência de infinitos pontos deve ser considerada apenas como uma interceptação a mais.

Inicialmente, consideramos que o número de intersecções é zero, e a cada nova intersecção encontrada, incrementamos em um este contador. Se o número final de intersecções contadas for par, o ponto será externo ao domínio, ao passo que, sendo ímpar, implica que ele é interno ao domínio.

Portanto, com tal procedimento, conseguimos estabelecer os quadrantes internos e externos ao domínio, finalizando o processo de criação da malha quadtree a partir da árvore quaternária.

Neste momento, como já é de praxe, recomendamos que o leitor verifique no código fonte do *QuadMesh2D* a implementação dos algoritmos descritos nesta seção, bem como utilize o mesmo para a confecção de alguns testes que lhe permita melhor compreender a técnica de geração de malhas quadtree.

Capítulo 3

Resultados

3.1 Introdução

Uma vez apresentados os conceitos e a técnica que proporciona a criação de malhas quadtree, aproveitamos este capítulo para apresentar resultados obtidos com a implementação desenvolvida.

A exibição desses resultados tem como objetivo primeiro a ilustração da técnica até então esboçada (sendo que, para isto, apresentamos a saída de cada uma das etapas mencionadas, as quais "particionam" a metodologia de criação da malha quadtree). Tal propósito é manifestado, principalmente, através dos problemas 1 e 2, os quais são bastante simples.

Já em um segundo momento, faz-se conveniente exemplificar todos os recursos disponíveis no programa no que diz respeito às possibilidades de definição do domínio e poligonais de controle, bem como exemplificar a geração de malhas quadtree em um domínio mais "geral"que os dois anteriores. Assim, com tal intuito, mostramos o problema 3.

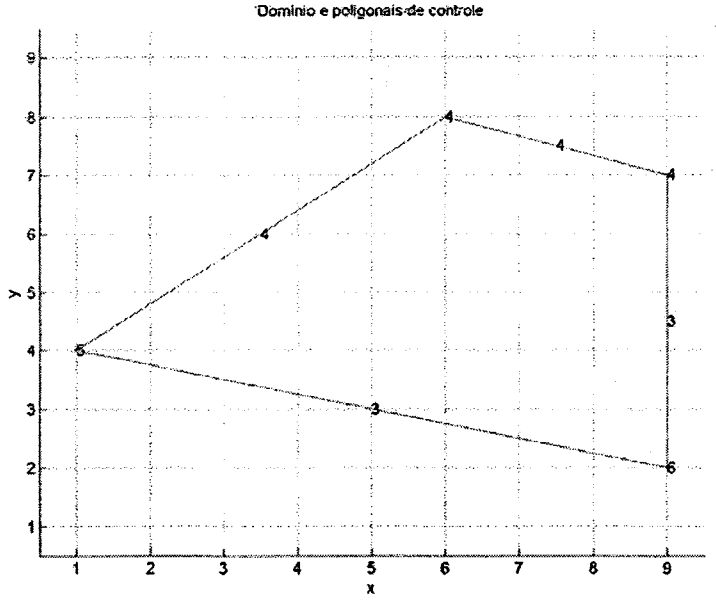
Nos demais problemas, a geração de uma malha quadtree objetiva mostrar a flexibilidade do programa na geração de diferentes malhas em domínios bem mais complicados que os anteriores, os quais necessitam de um número bem maior de pontos para serem definidos.

De modo geral, todos os exemplos dados contribuem para a verificação da corretude da implementação construída.

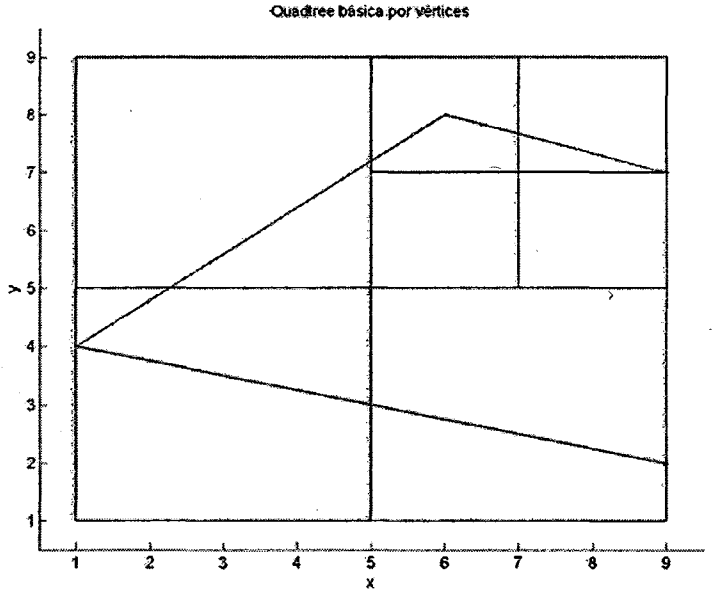
Antes de passarmos aos resultados propriamente ditos, salientamos que linhas vermelhas sempre demarcam a fronteira do domínio, enquanto que linhas verdes estão associadas a poligonais de controle e linhas azuis fazem menção ou aos lados dos quadrantes ou a elementos da malha quadtree. Além disso, números sobre o contorno do domínio ou sobre uma poligonal de controle remetem aos níveis mínimos de refinamento exigido pelo respectivo ente geométrico associado. Por fim, em todos os exemplos, a fim de melhor ilustrarmos o processo de balanceamento local, o desnível máximo permitido para quadrantes vizinhos foi igual a um, muito embora o *QuadMesh2D* possa operar normalmente com qualquer tolerância que vier a ser estabelecida.

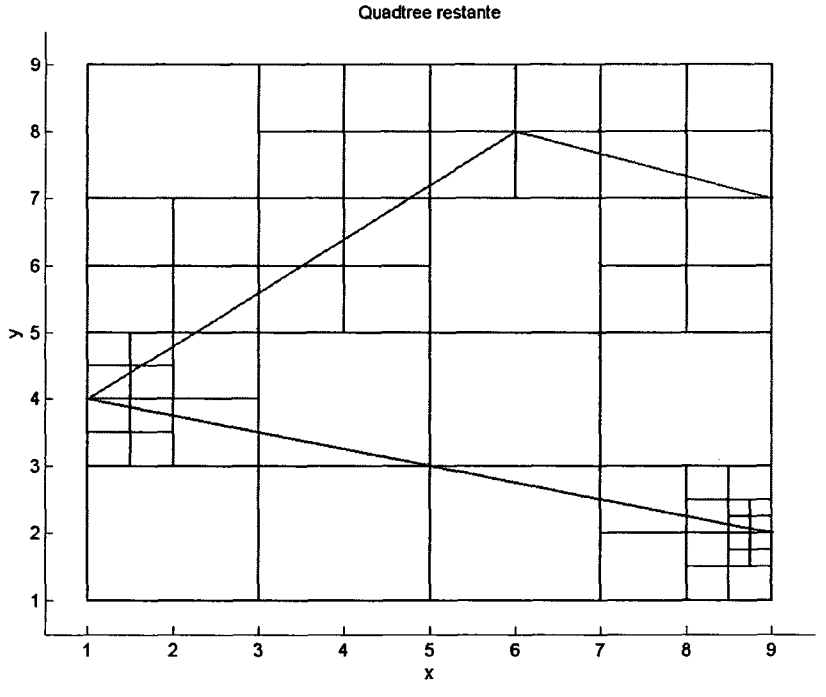
3.2 Problema 1

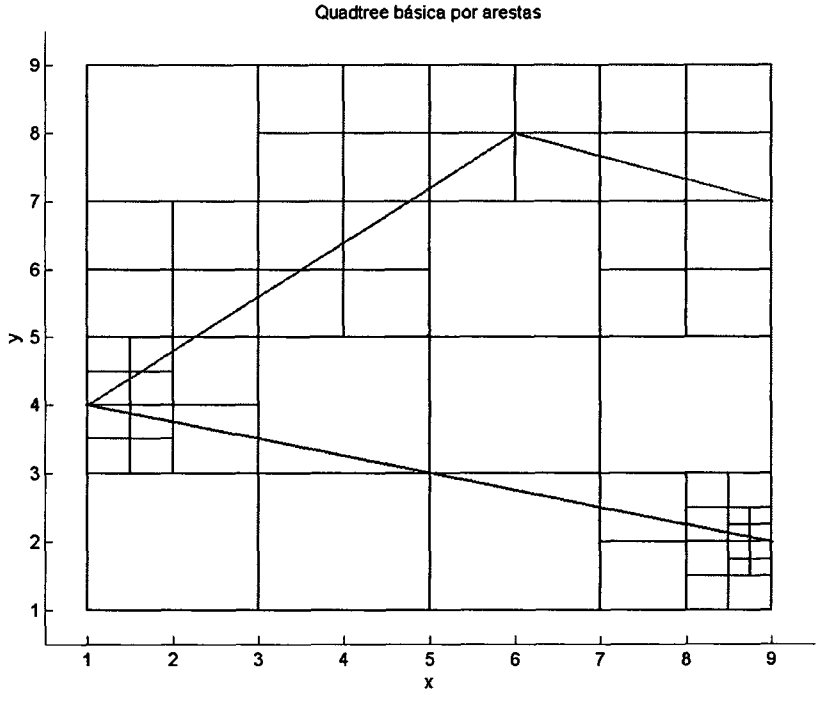
Aqui, consideramos a geração da malha quadtree no interior do polígono representado na figura abaixo,

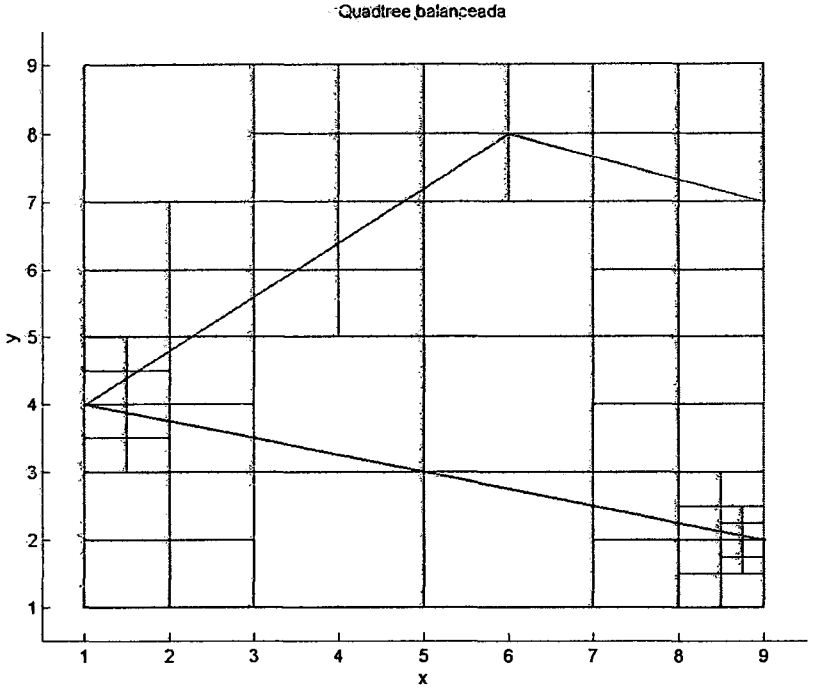


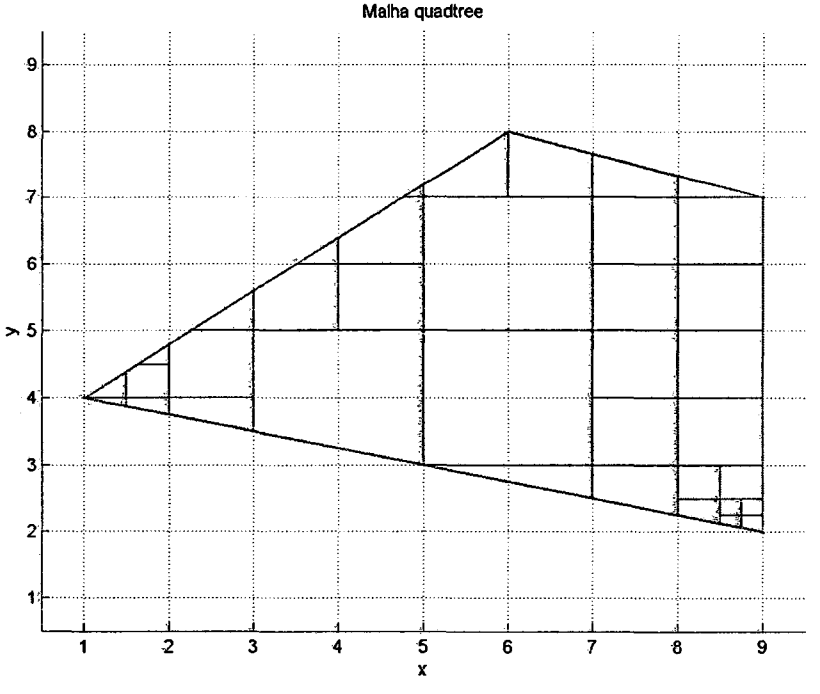
com todas as etapas de geração da malha sendo ilustradas, sequencialmente, no que segue.




Quadtree restante
y
x
1
2
3
4
5
6
7
8
9

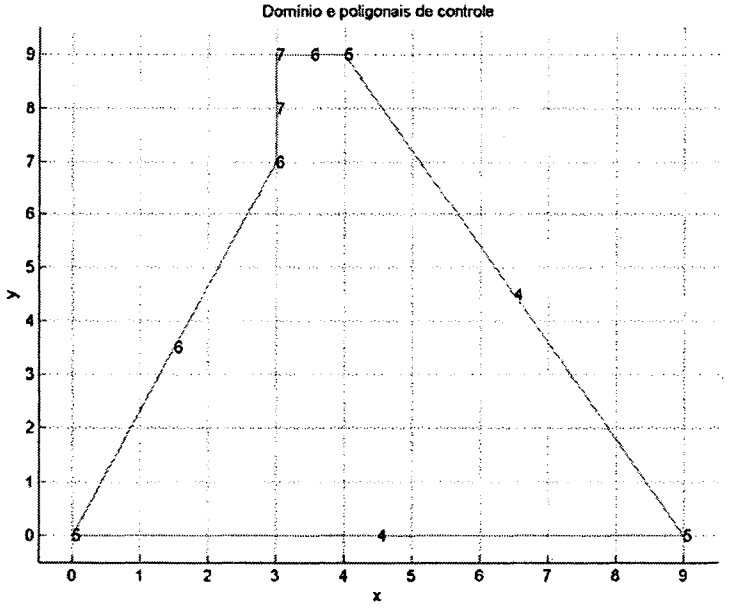

Quadtree básica por arestas
y
x
1
2
3
4
5
6
7
8
9


Quadtree balanceada
y
x
1
2
3
4
5
6
7
8
9

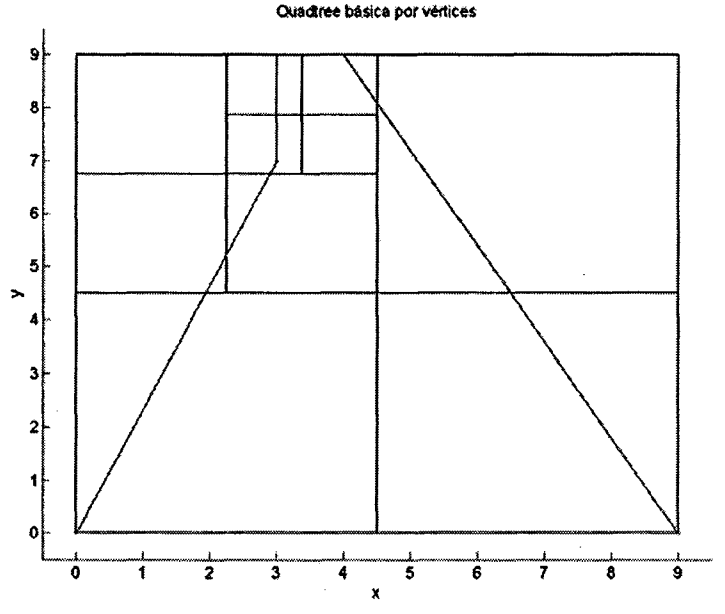

Malha quadtree
y
x
1
2
3
4
5
6
7
8
9

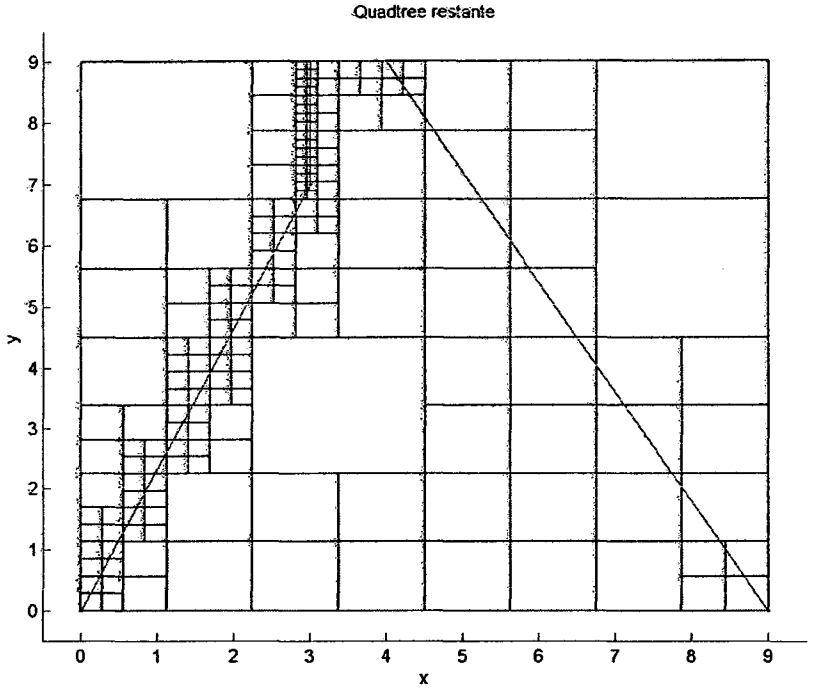
3.3 Problema 2

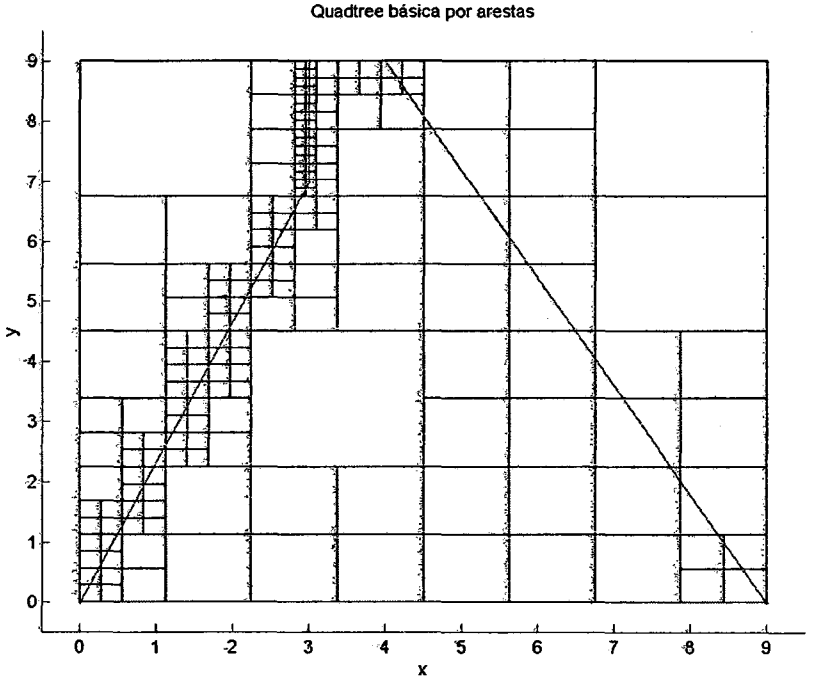
Para este caso, extraído de [ZC68], consideramos o modelo de uma barragem de gravidade, dada por

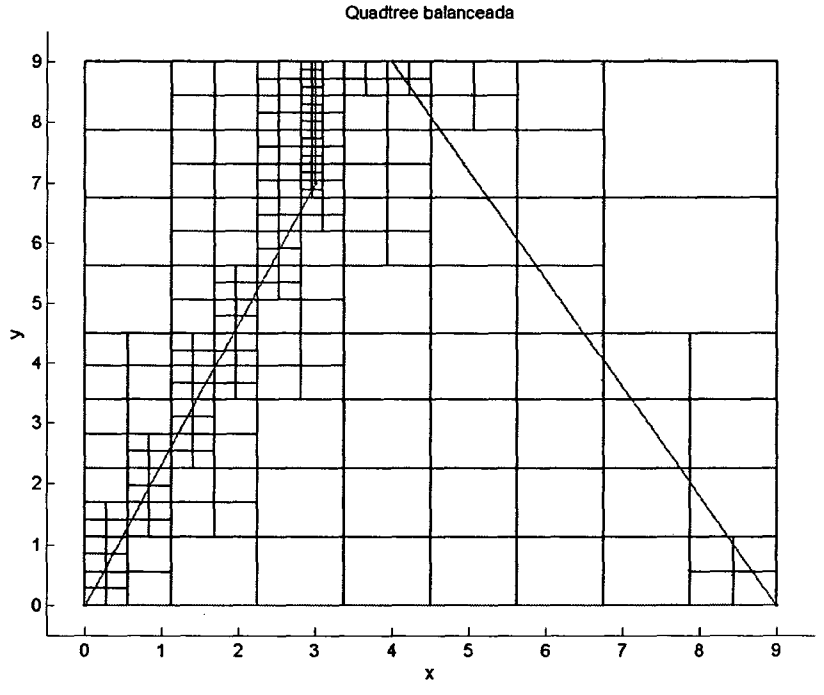


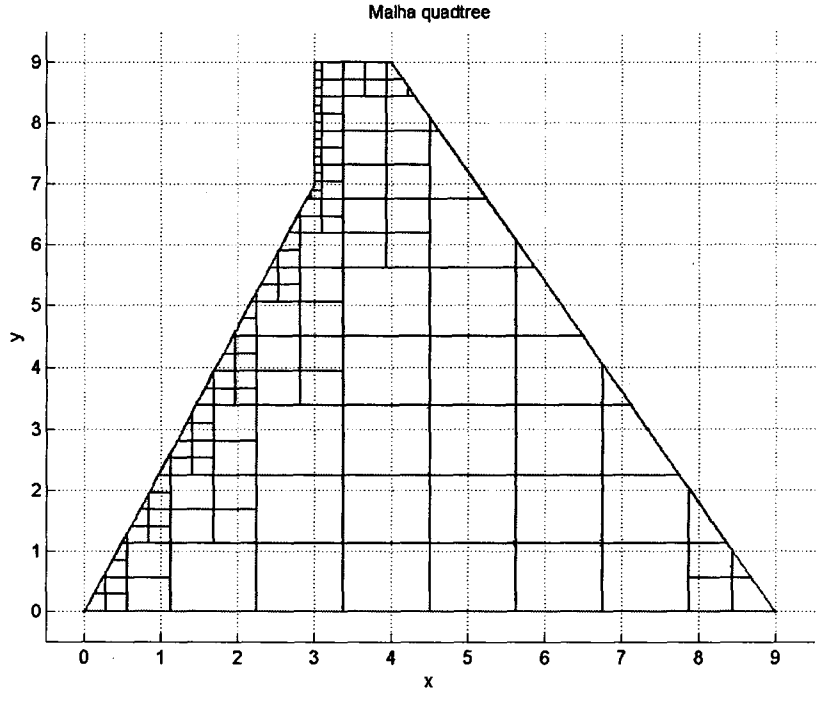
com todas as etapas de geração da malha sendo ilustradas, sequencialmente, no que segue.




Quadtree restante
y
x
0
1
2
3
4
5
6
7
8
9

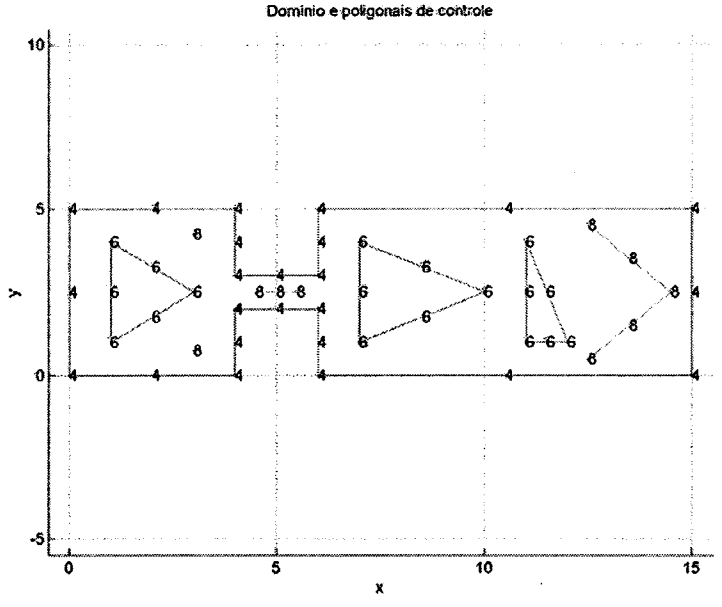

Quadtree básica por arestas
y
x
0
1
2
3
4
5
6
7
8
9


Quadtree balanceada
y
x
0
1
2
3
4
5
6
7
8
9

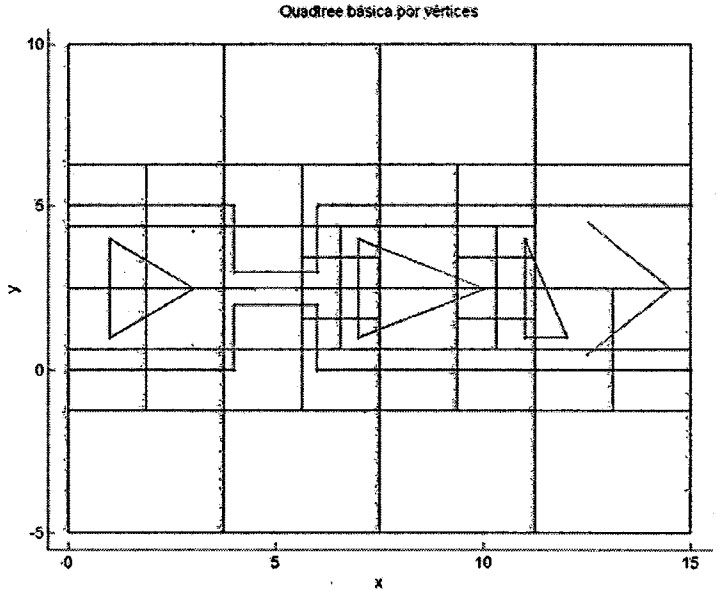

Malha quadtree
y
x
0
1
2
3
4
5
6
7
8
9

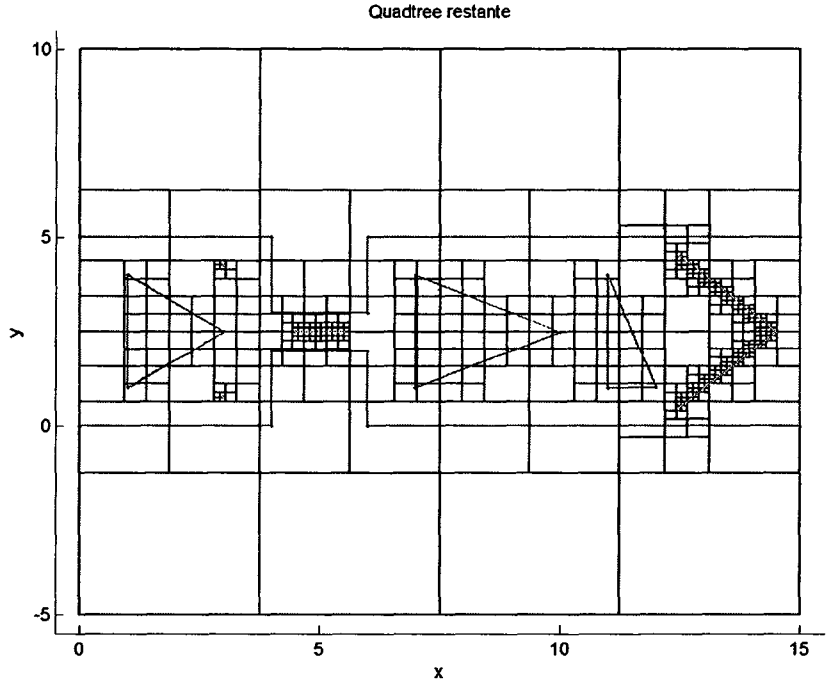
3.4 Problema 3

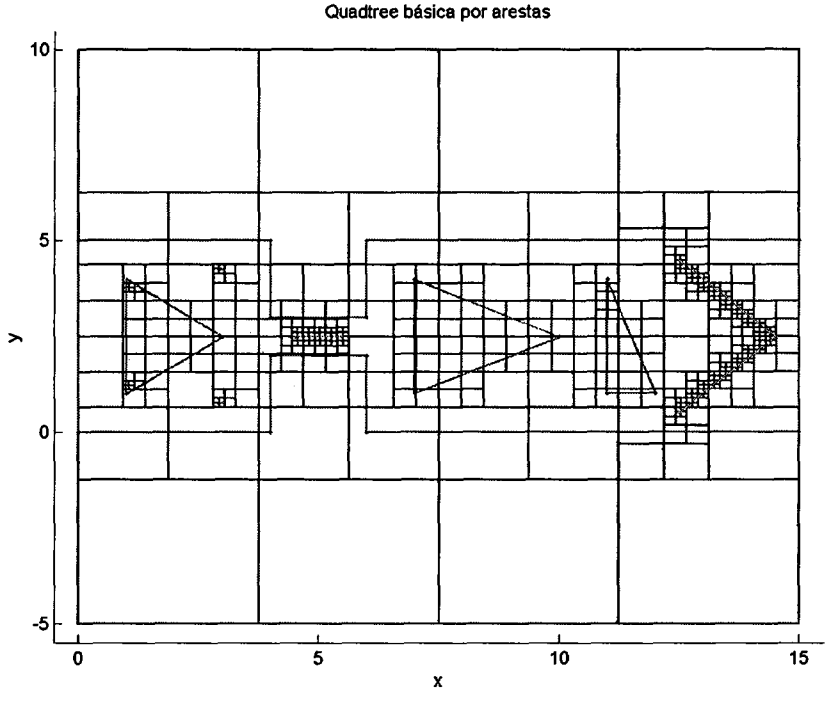
Consideramos agora a malha gerada no interior de um canal com obstáculos internos e algumas poligonais e pontos de controle, isto é,

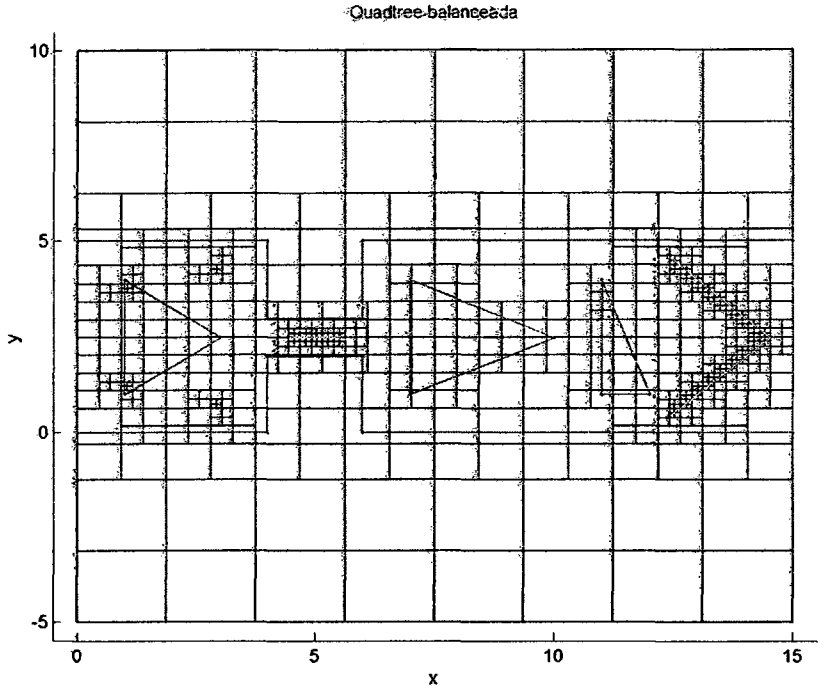


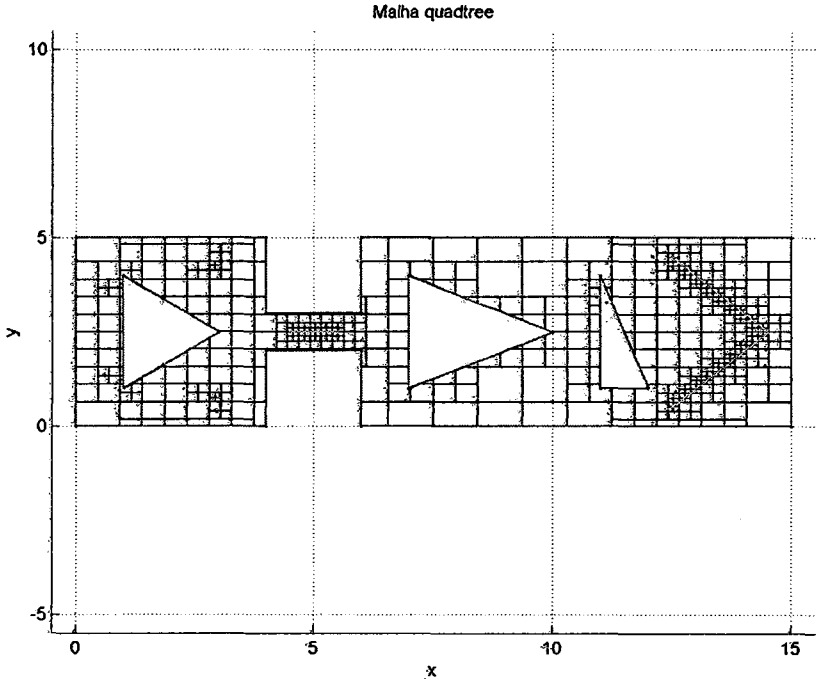
com todas as etapas de geração da malha sendo ilustradas, sequencialmente, no que segue.




Quadtree restante
10
5
0
-5
y
0
5
10
15
x

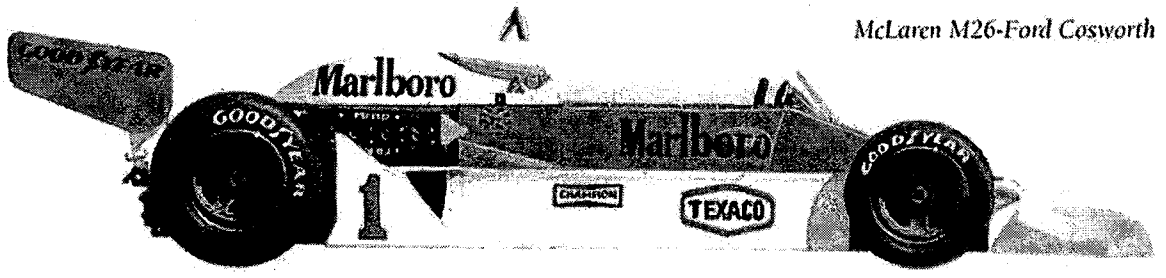

Quadtree básica por arestas
10
5
0
-5
y
0
5
10
15
x


Quadtree-balanceada
10
5
0
-5
y
0
5
10
15
x


Malha quadtree
10
5
0
-5
y
0
5
10
15
x

3.5 Problemas 4, 5, 6 e 7

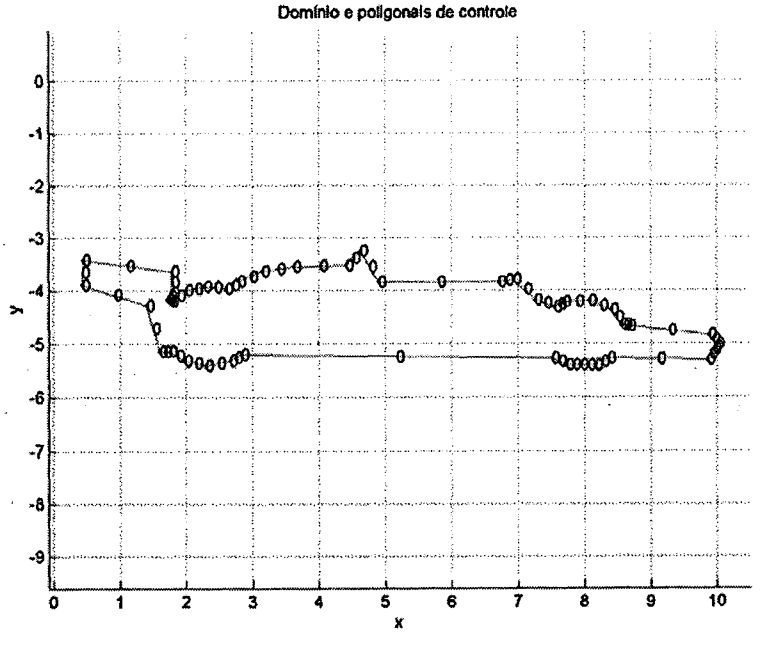
Nesta sequência de problemas, consideramos o contorno externo associado à seguinte imagem,



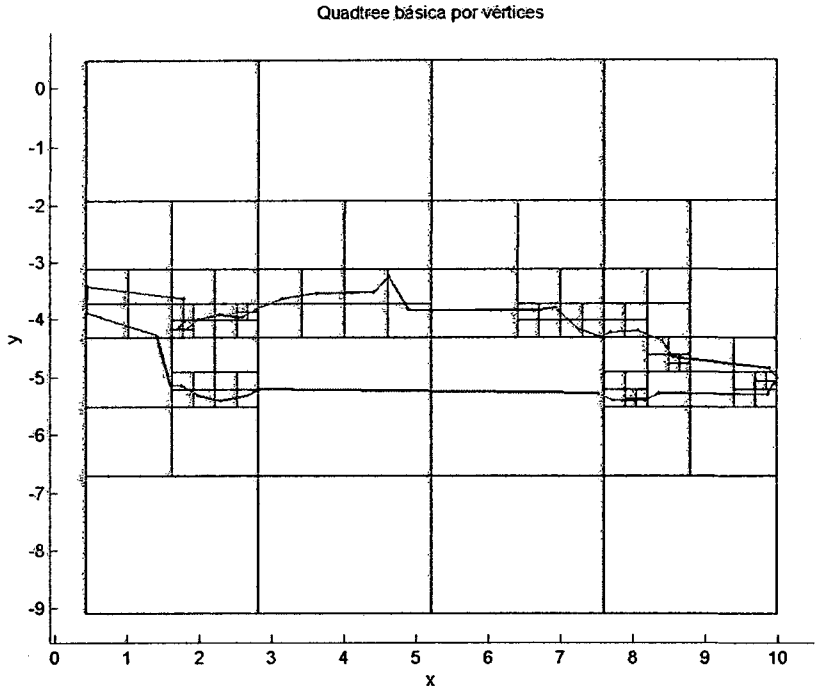
que corresponde ao modelo guiado por James Hunt no GP Britânico de F1 do ano de 1977.

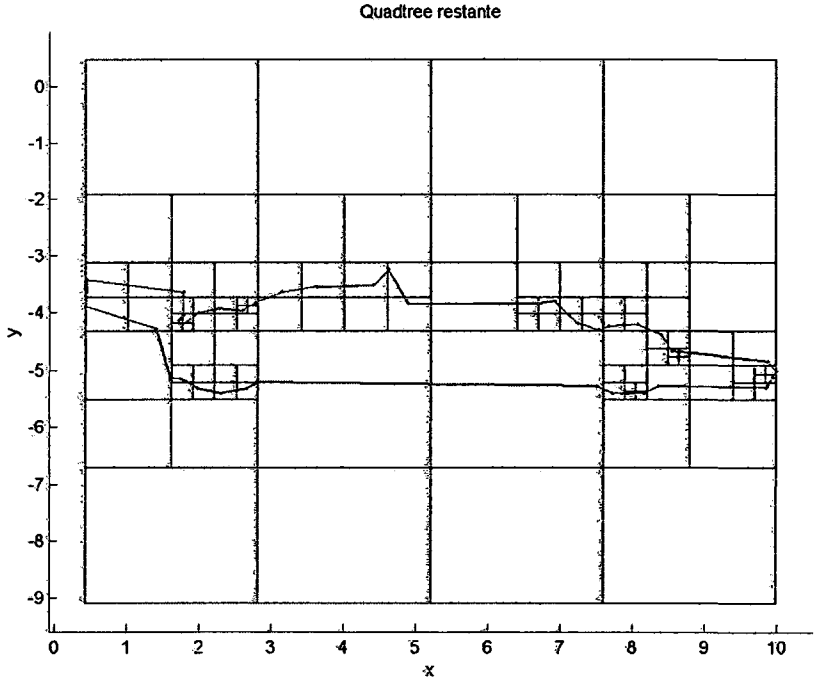
Assim, inicialmente, a fim de verificarmos que a quadtree básica é suficiente para representar o domínio, tendo em vista a geração da malha quadtree, os problemas 4 e 5 adotarão um nível mínimo de refinamento nulo em todas as porções do domínio.

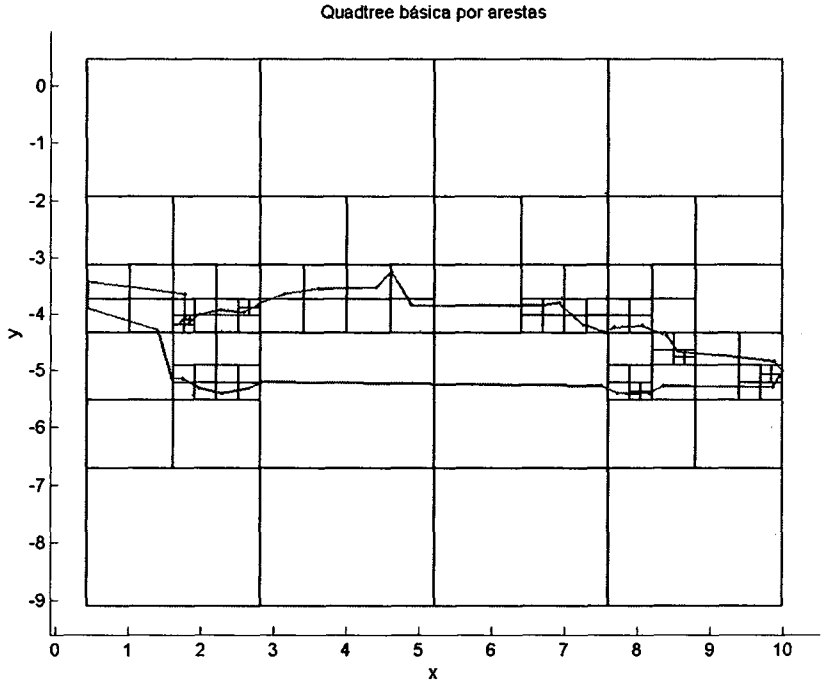
Dessa forma, no problema 4, pretendemos gerar uma malha interna ao ciclo existente, isto é,

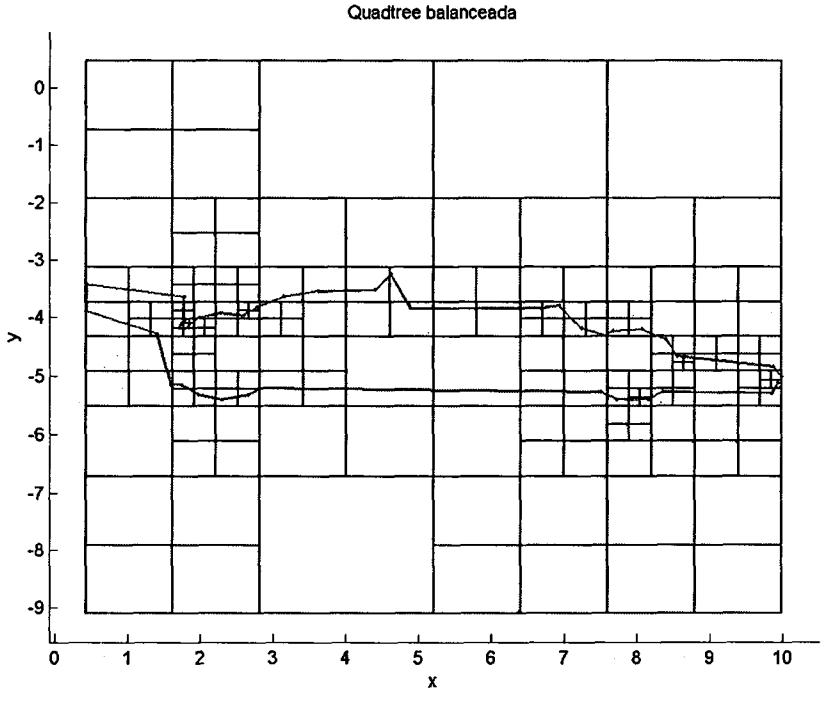


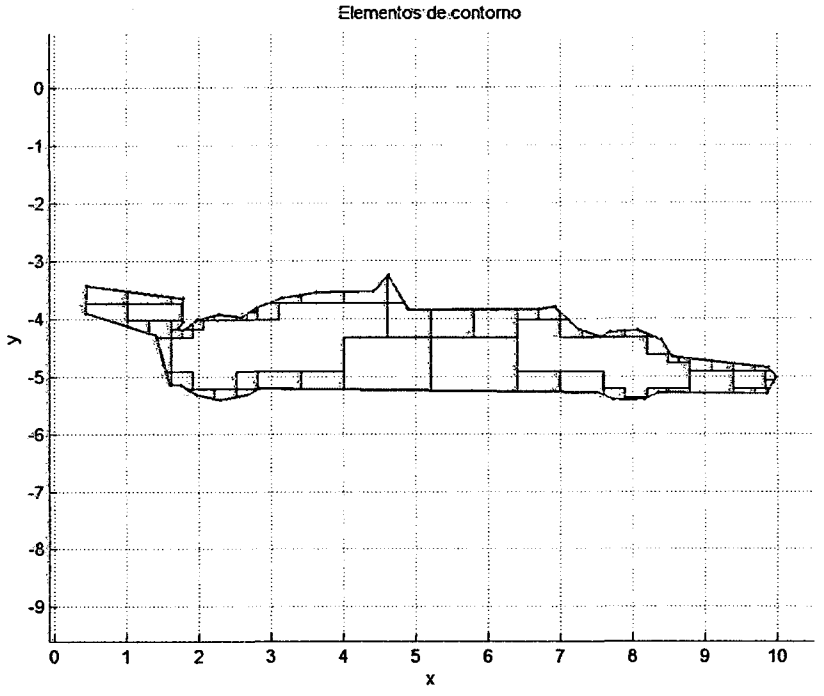
com todas as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.

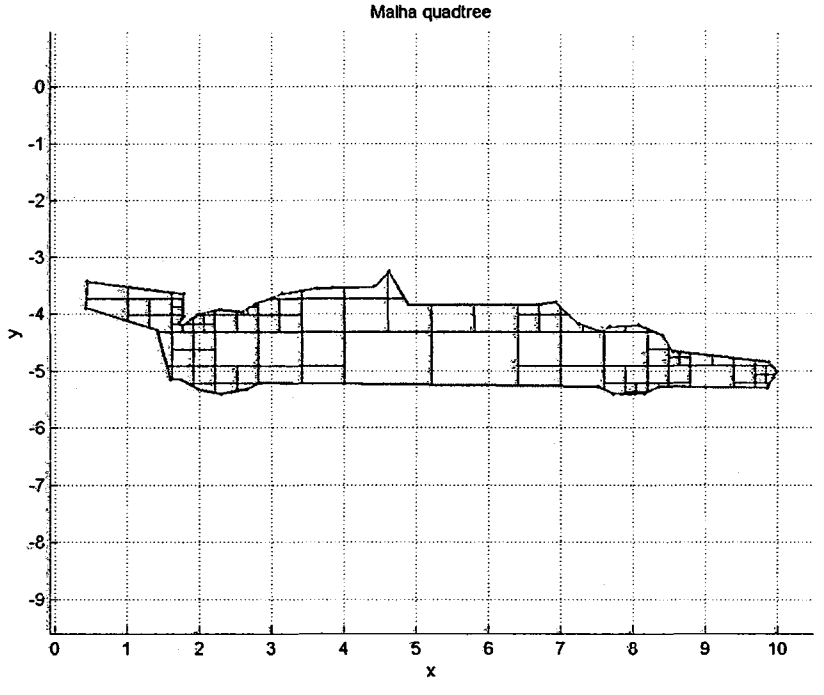

Quadtree básica por vértices
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
y
0 1 2 3 4 5 6 7 8 9 10
x


Quadtree restante
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
y
0 1 2 3 4 5 6 7 8 9 10
x

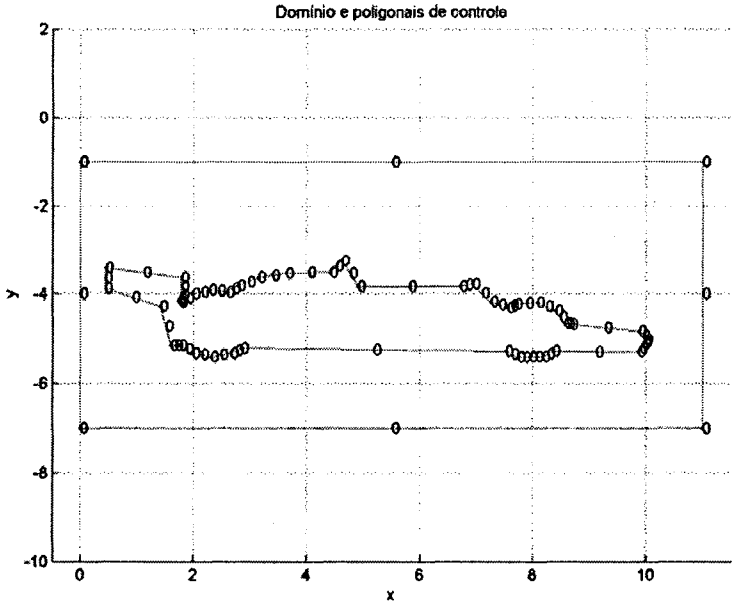

Quadtree básica por arestas
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
y
0 1 2 3 4 5 6 7 8 9 10
x


Quadtree balanceada
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
y
0 1 2 3 4 5 6 7 8 9 10
x

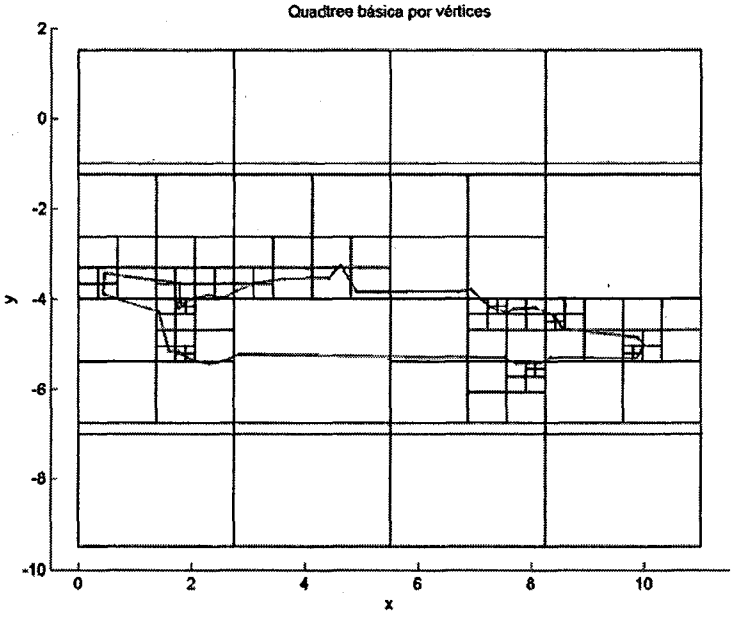

Elementos de contorno
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
y
0 1 2 3 4 5 6 7 8 9 10
x

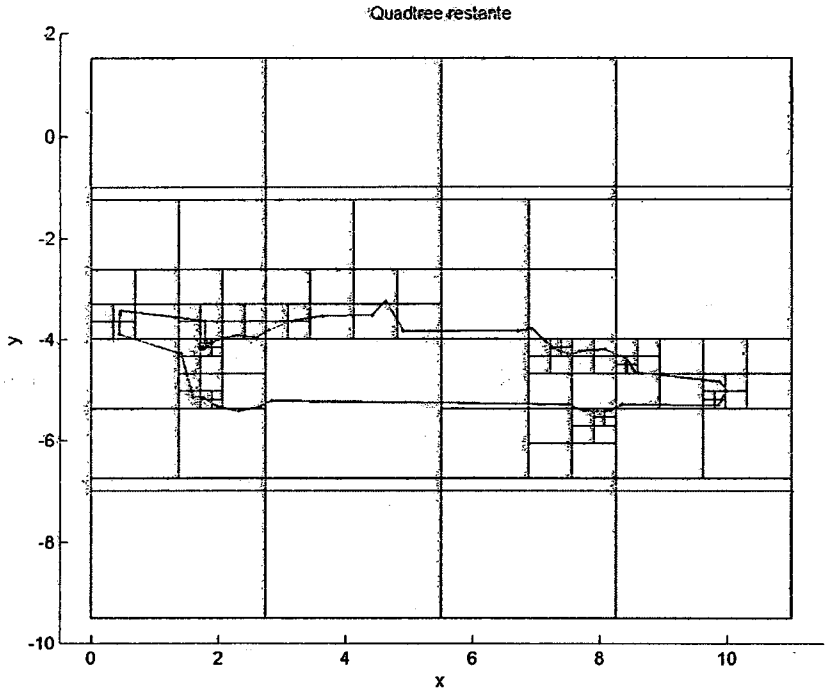

Malha quadtree
0
-1
-2
-3
-4
-5
-6
-7
-8
-9
y
0 1 2 3 4 5 6 7 8 9 10
x

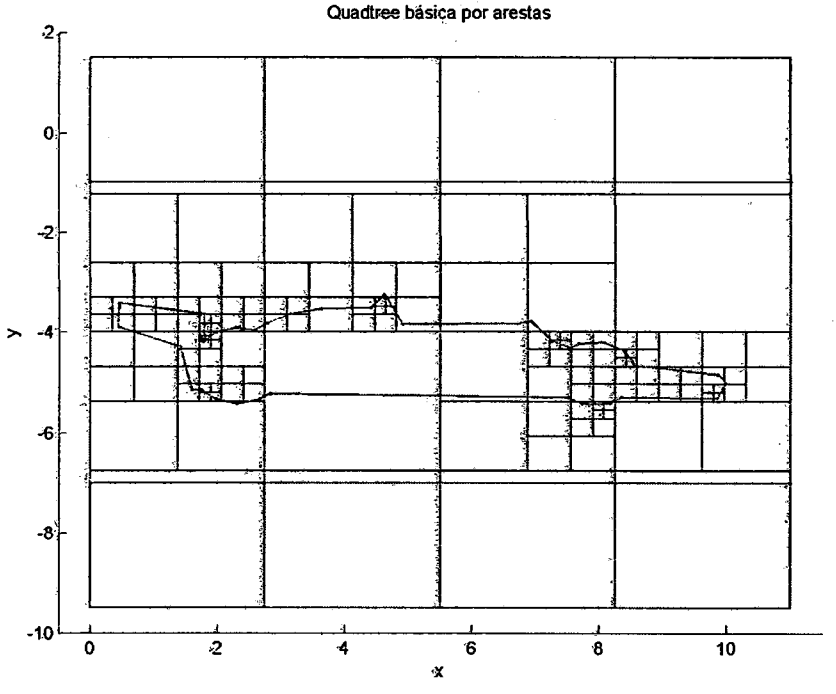
Agora, já no problema 5, passamos a usar o contorno considerado definindo um ciclo interno a um retângulo, isto é,

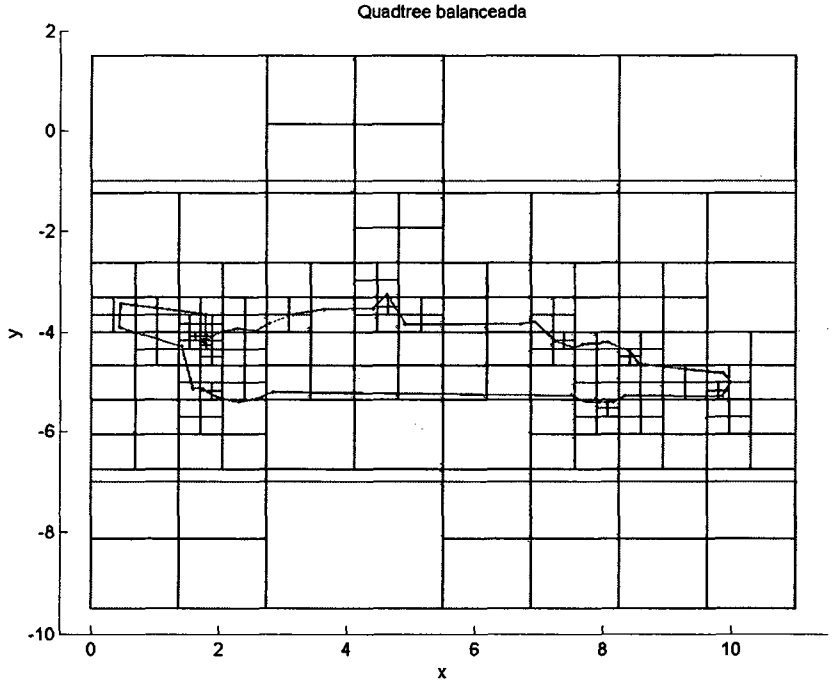


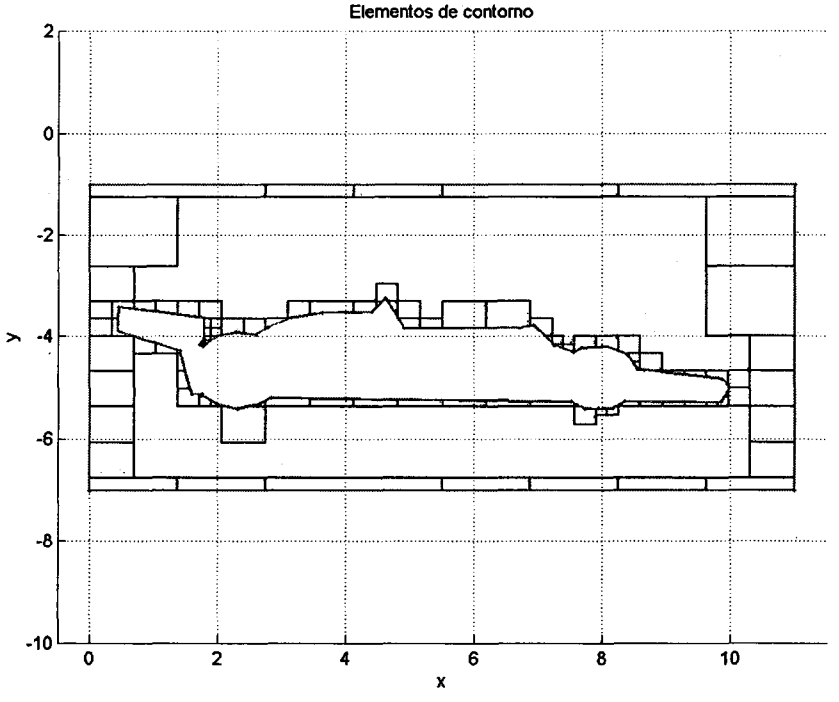
com todas as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.

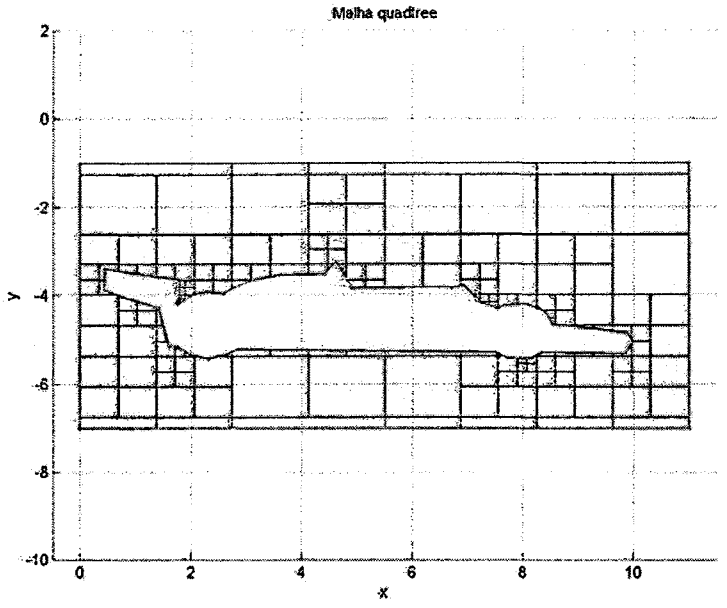



Quadtree restante
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

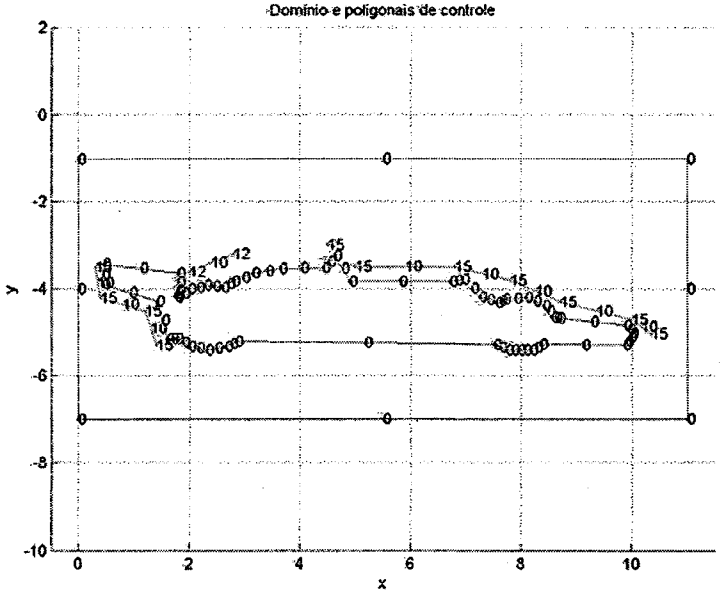

Quadtree básica por arestas
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x


Quadtree balanceada
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

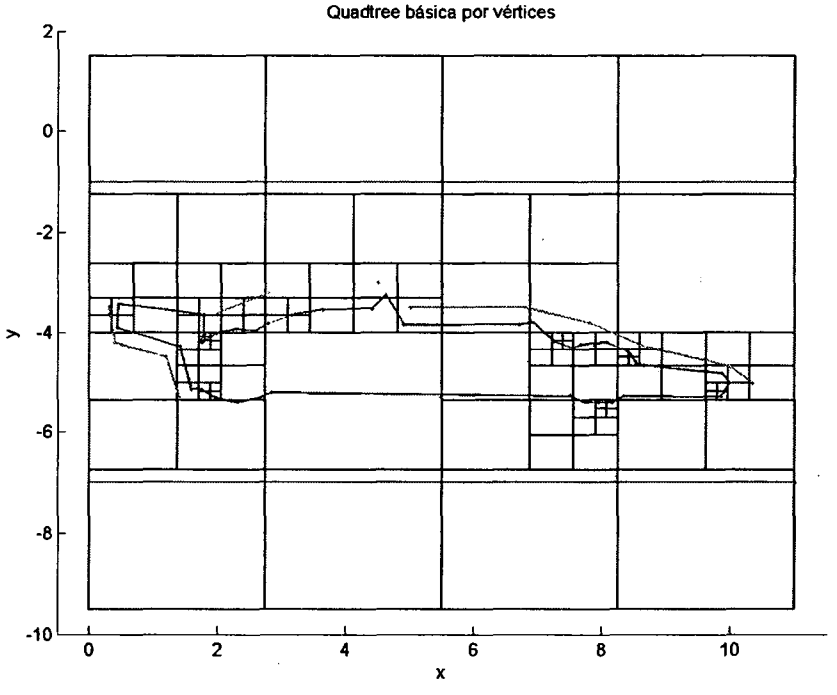

Elementos de contorno
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

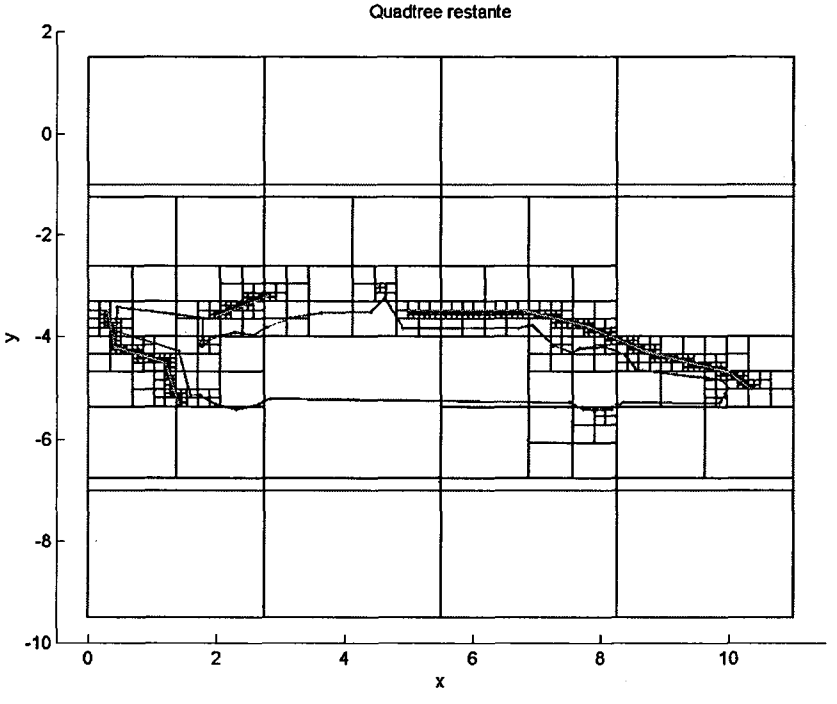


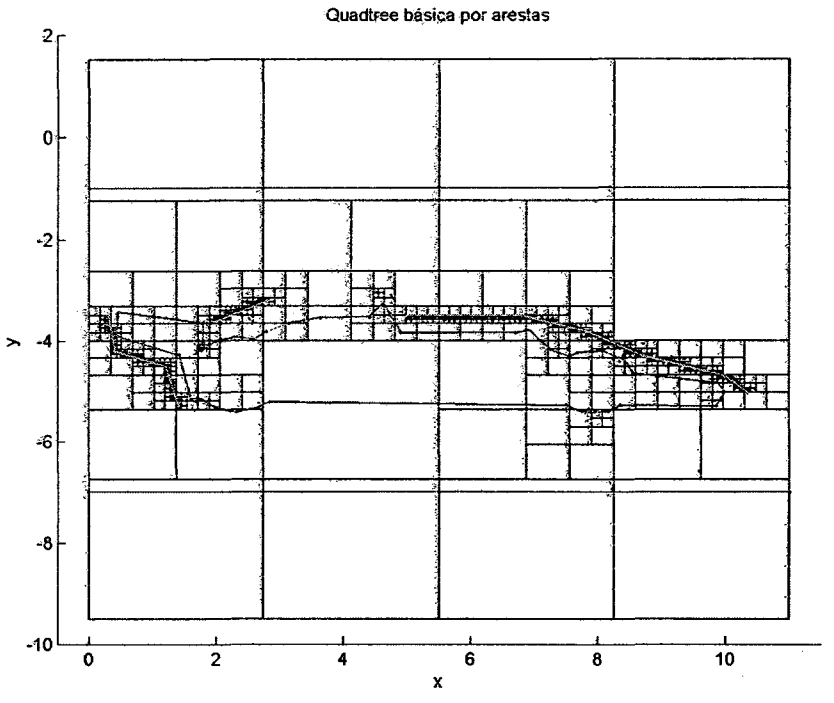
Para o problema 6, voltamos a considerar o domínio discretizado no problema 5, mas, com a inserção de algumas poligonais de controle, isto é,

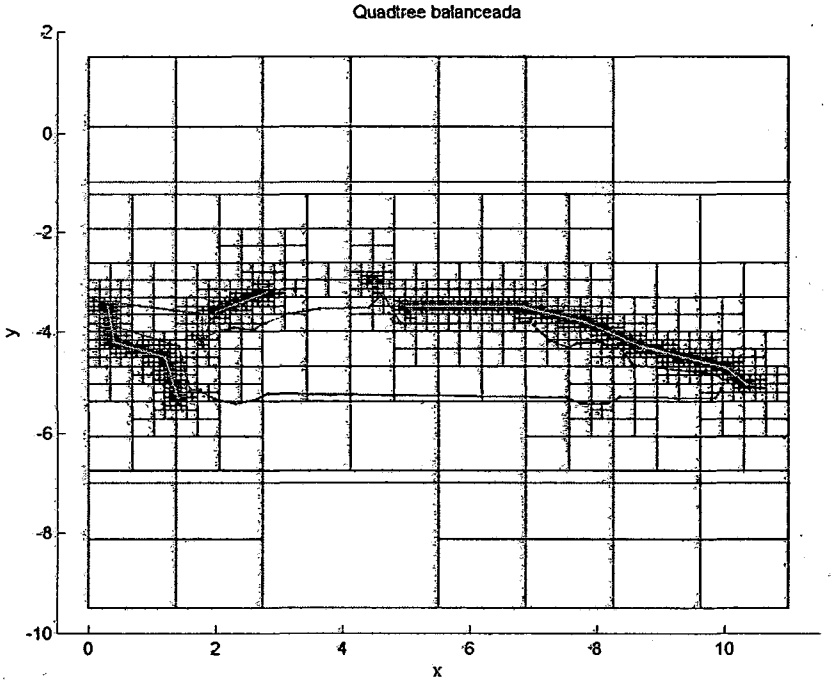


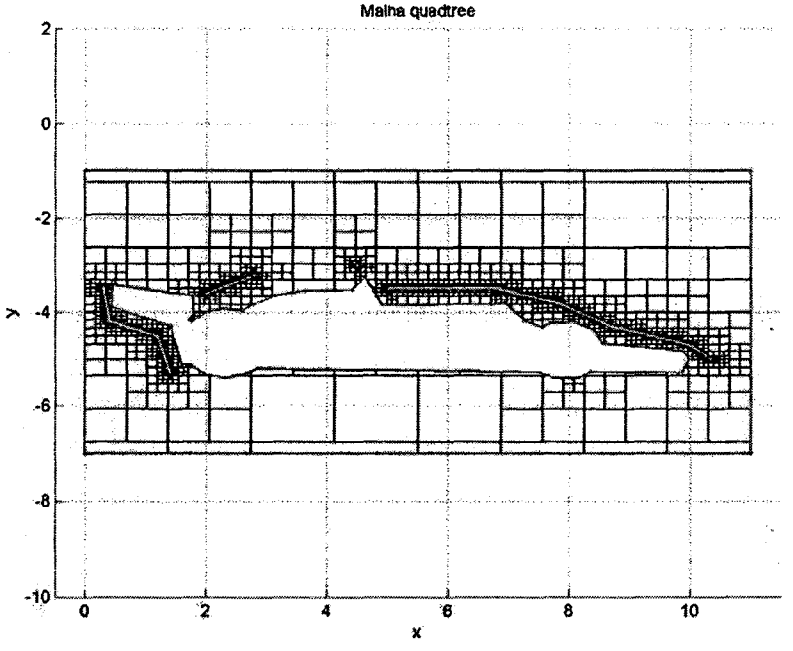
com todas as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.


Quadtree básica por vértices
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

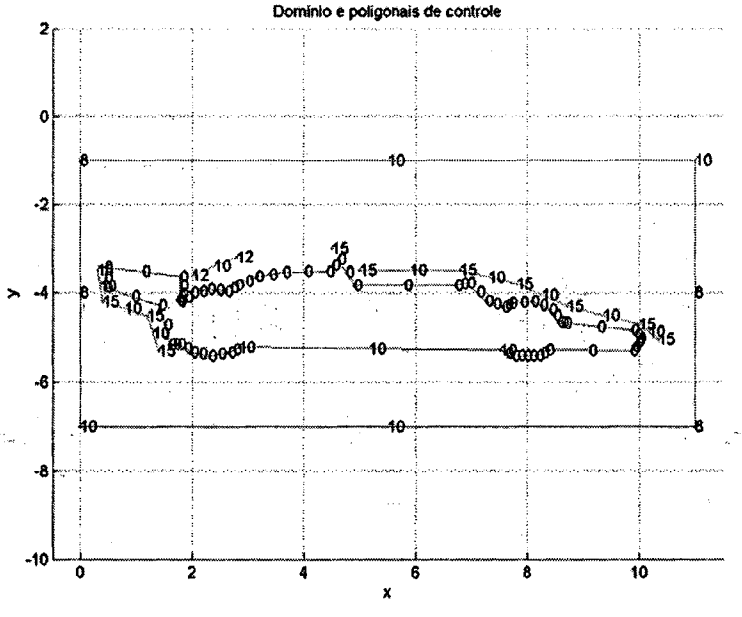

Quadtree restante
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x


Quadtree básica por arestas
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

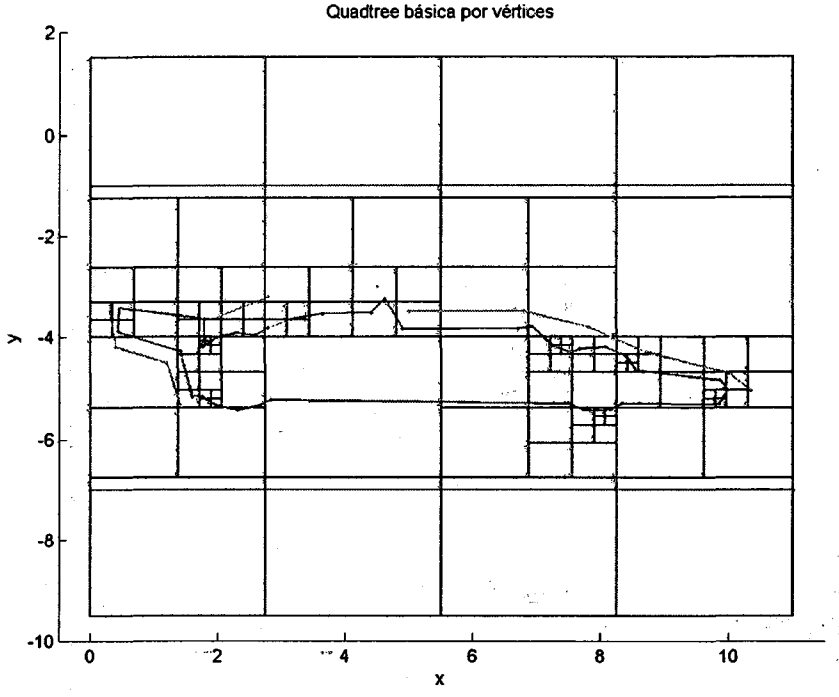

Quadtree balanceada
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

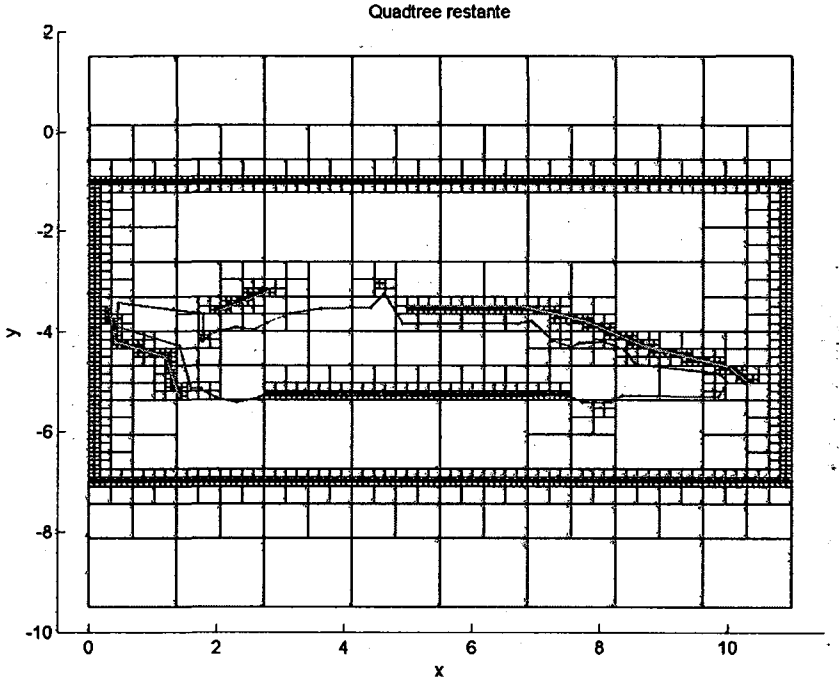


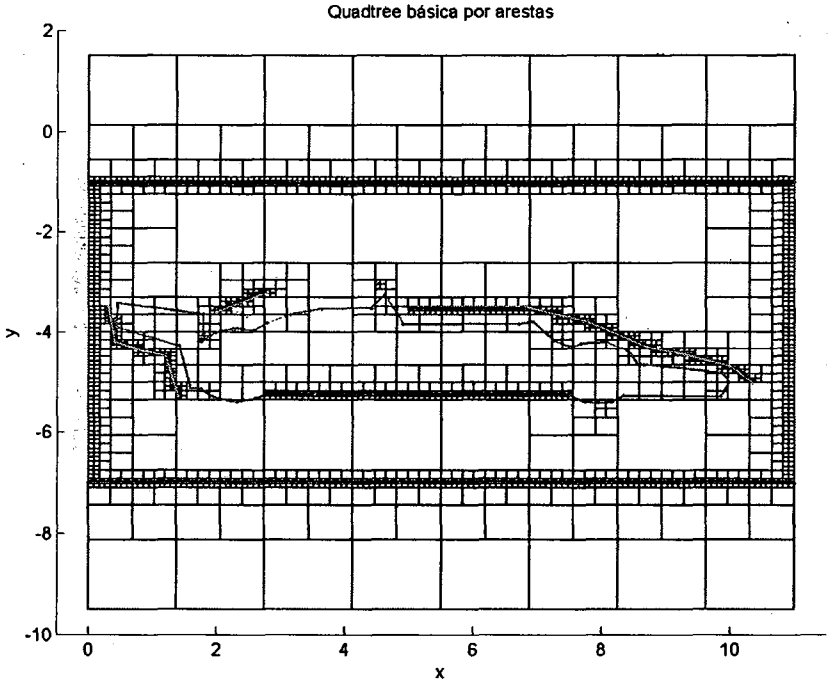
Finalmente, utilizamos este contorno pela última vez, adicionando alguns níveis mínimo de refinamento ao retângulo externo e ao assoalho do carro, isto é,

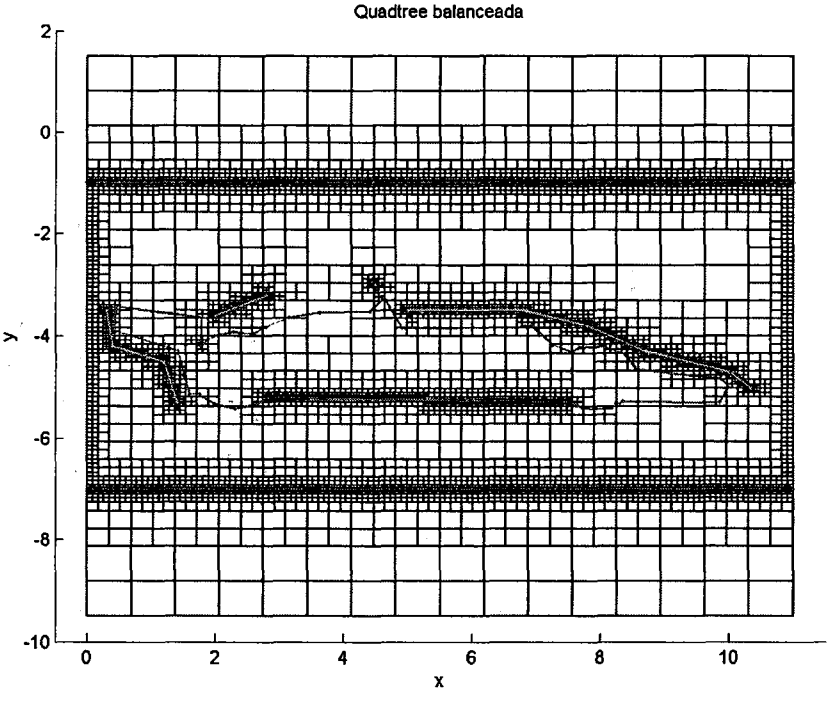


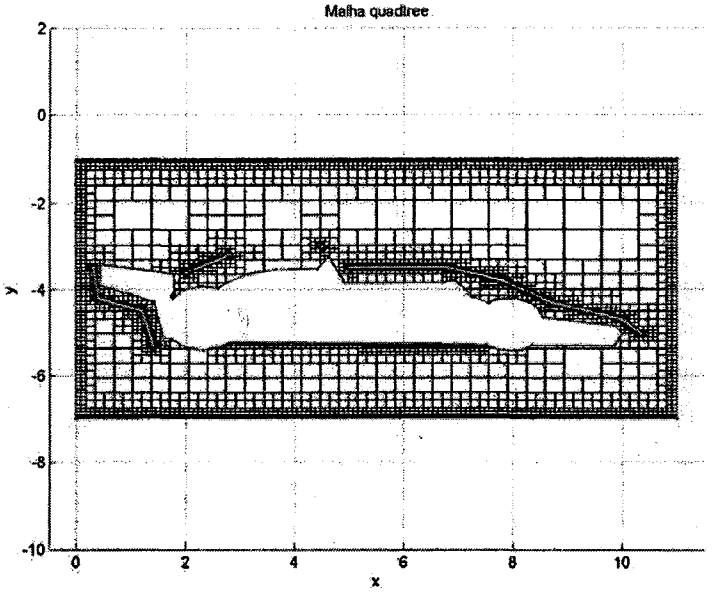
com todas as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.


Quadtree básica por vértices
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x


Quadtree restante
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

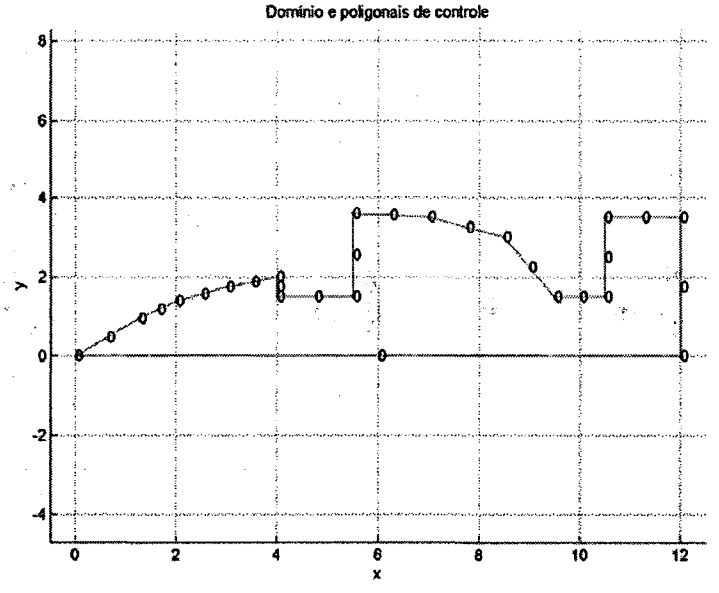

Quadtree básica por arestas
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x


Quadtree balanceada
2
0
-2
-4
-6
-8
-10
y
0
2
4
6
8
10
x

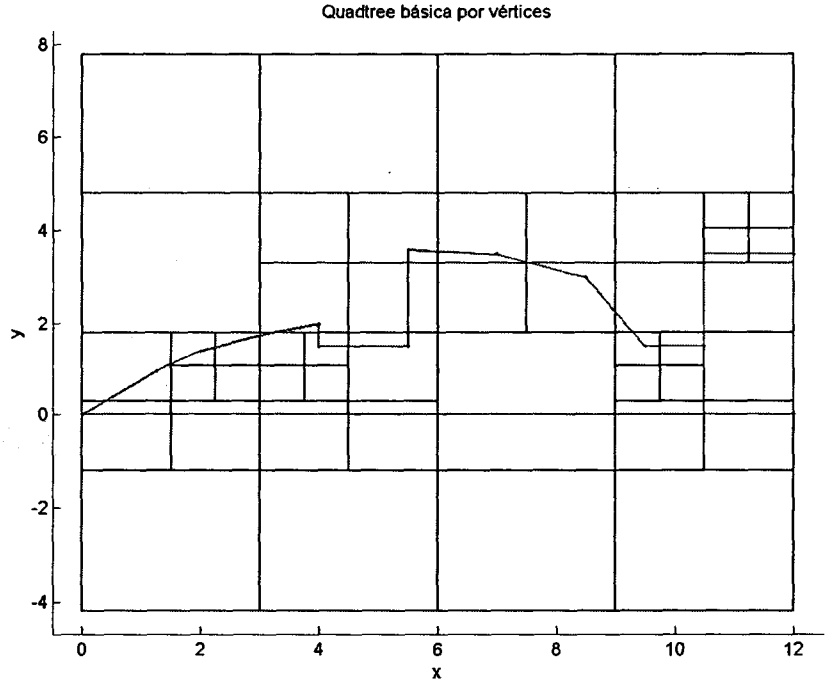


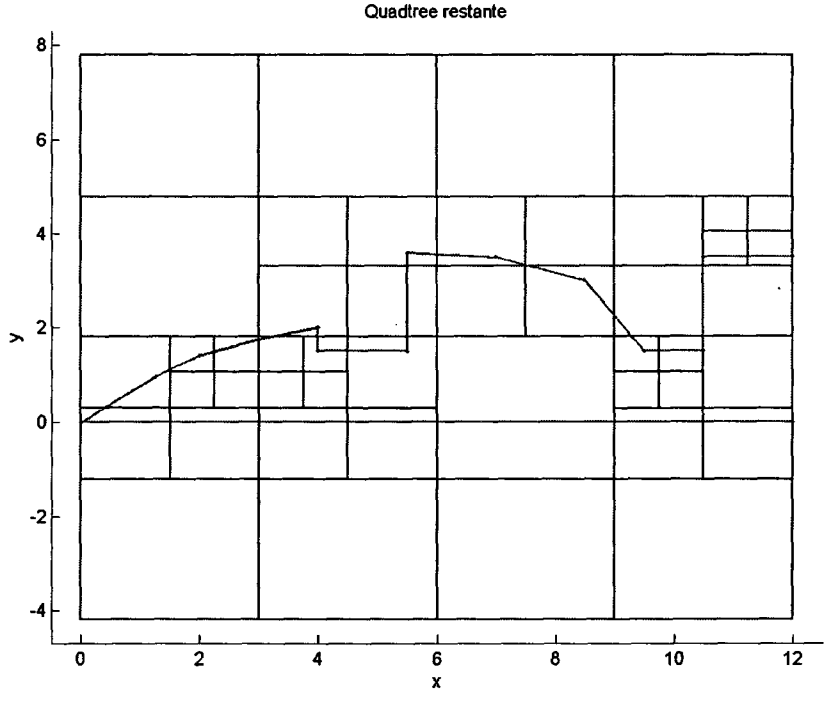
3.6 Problemas 8, 9 e 10

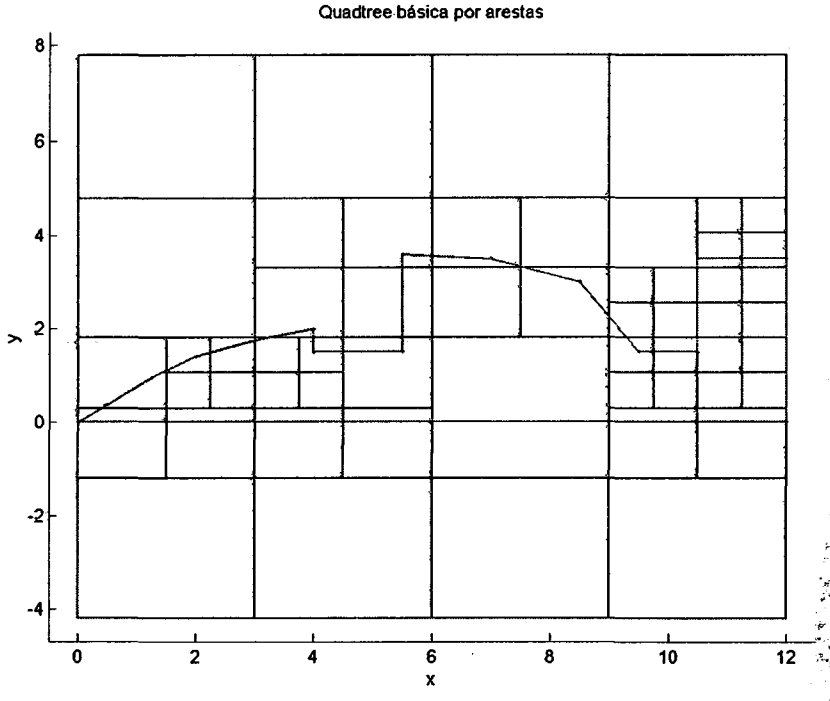
Consideraremos, agora, o seguinte domínio para o problema 8,

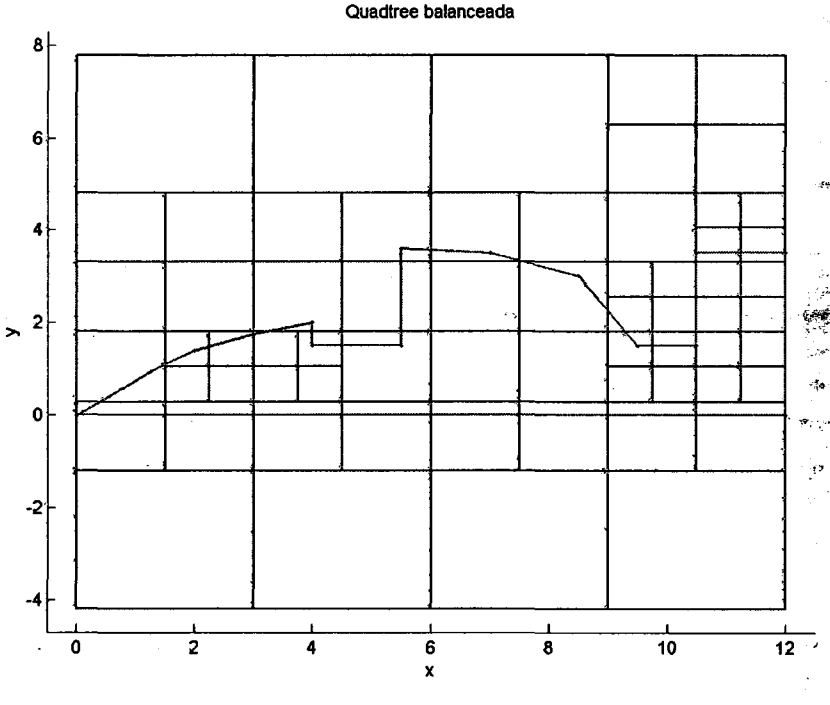


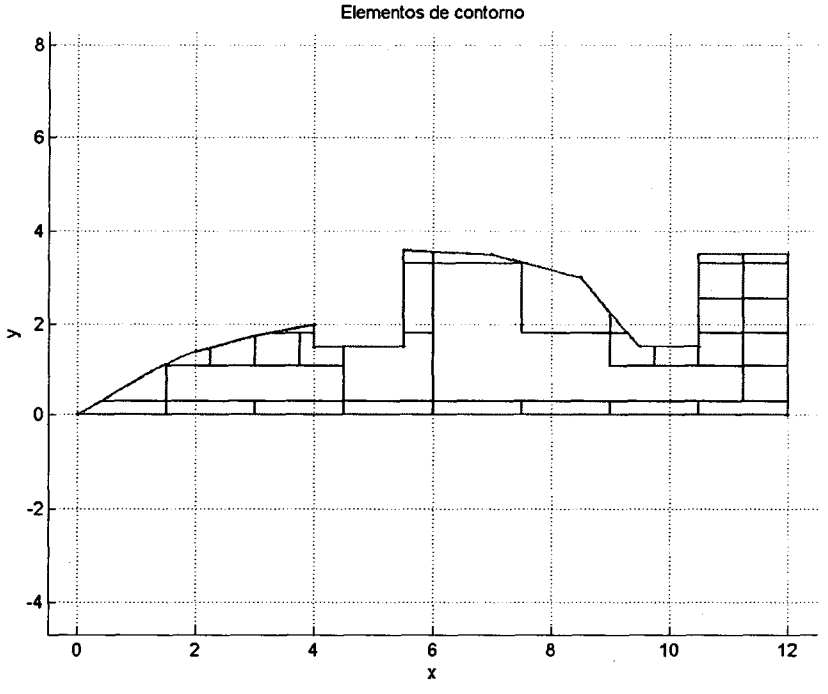
com todas as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.

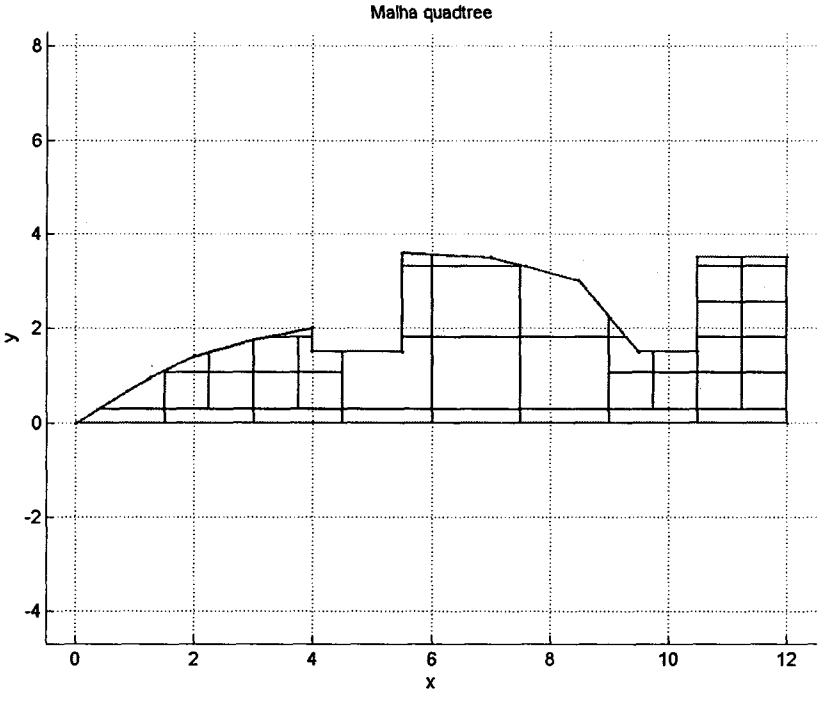

Quadtree básica por vértices
8
6
4
2
0
-2
-4
y
0
2
4
6
8
10
12
x


Quadtree restante
8
6
4
2
0
-2
-4
y
0
2
4
6
8
10
12
x

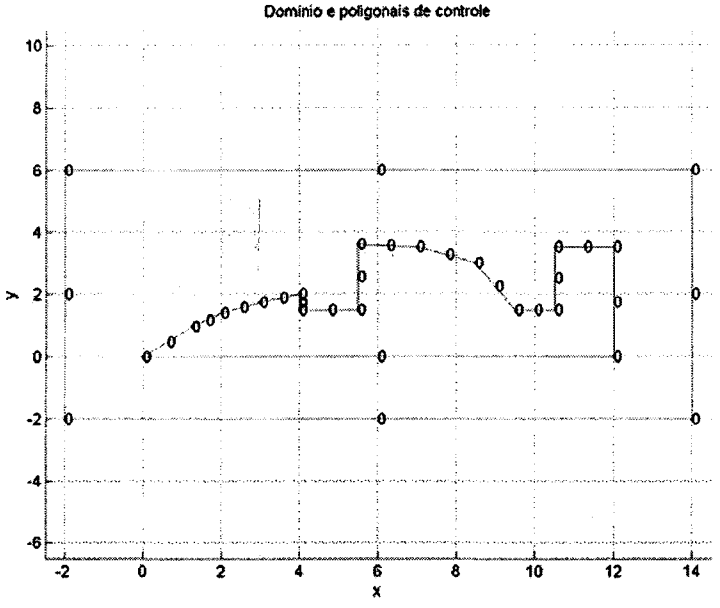

Quadtree básica por arestas
8
6
4
2
0
-2
-4
y
0
2
4
6
8
10
12
x


Quadtree balanceada
8
6
4
2
0
-2
-4
y
0
2
4
6
8
10
12
x

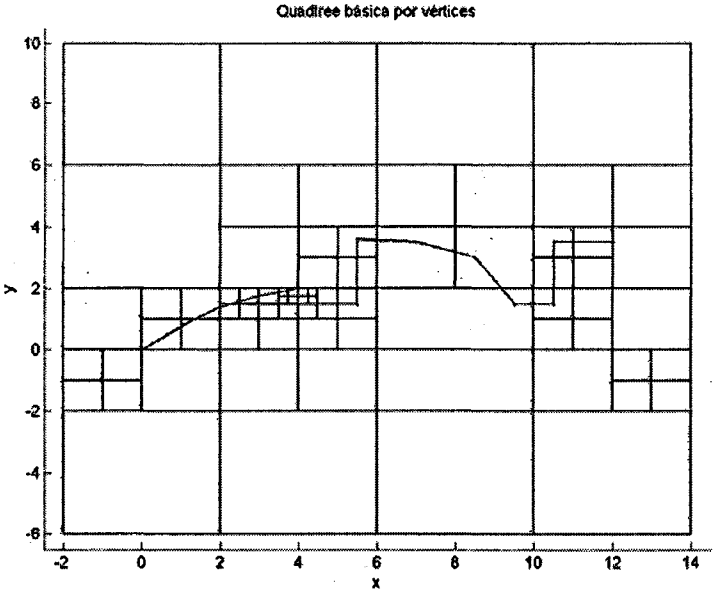

Elementos de contorno
8
6
4
2
0
-2
-4
y
0
2
4
6
8
10
12
x

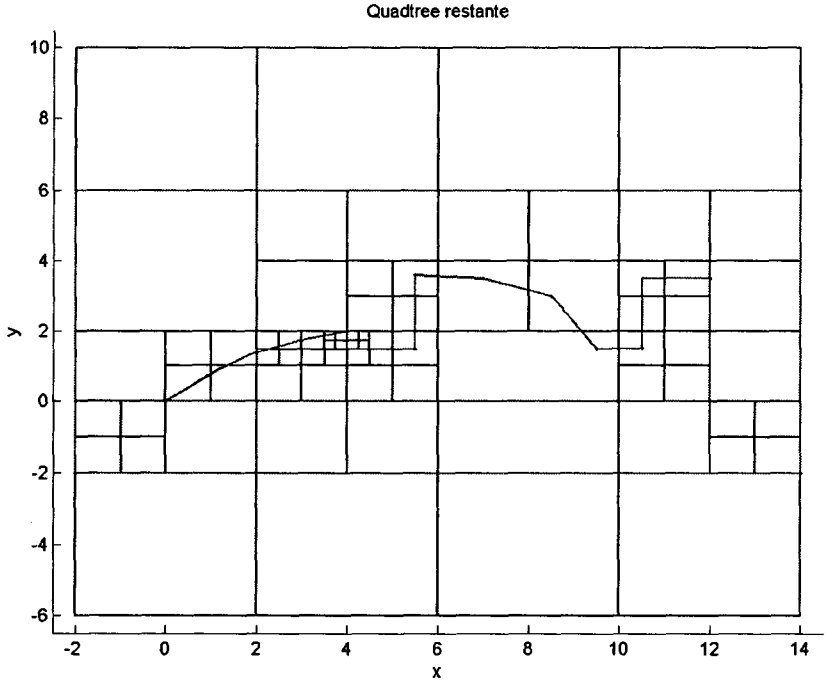

Malha quadtree
8
6
4
2
0
-2
-4
y
0
2
4
6
8
10
12
x

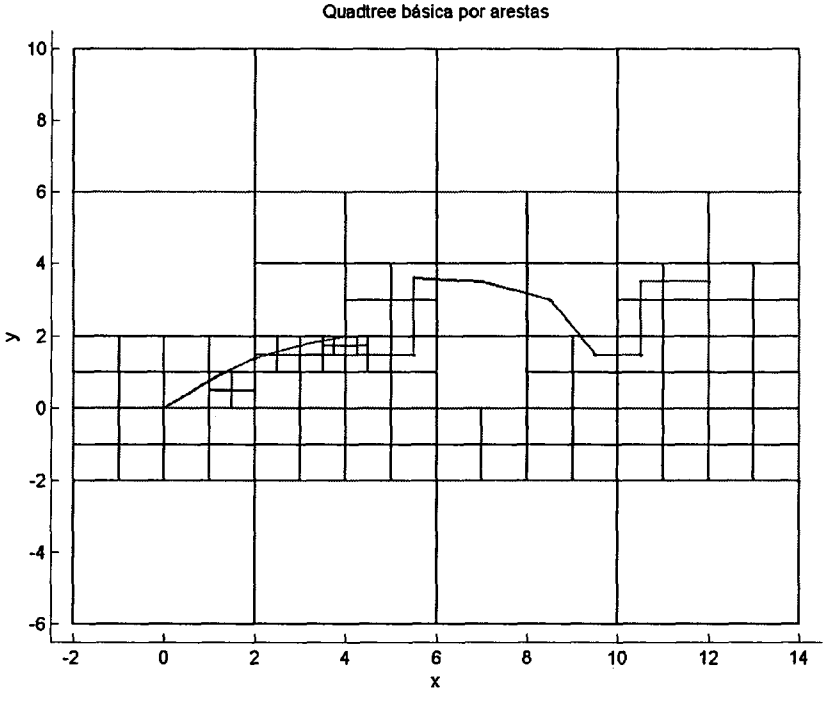
Aproveitando ainda este contorno como um contorno interno, trataremos, no problema 9, em gerar uma malha no domínio,

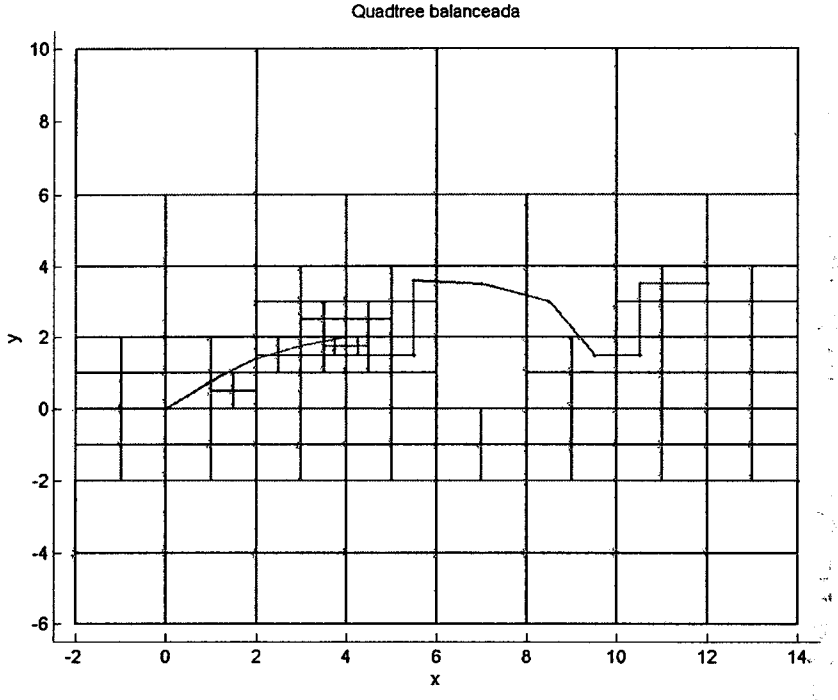


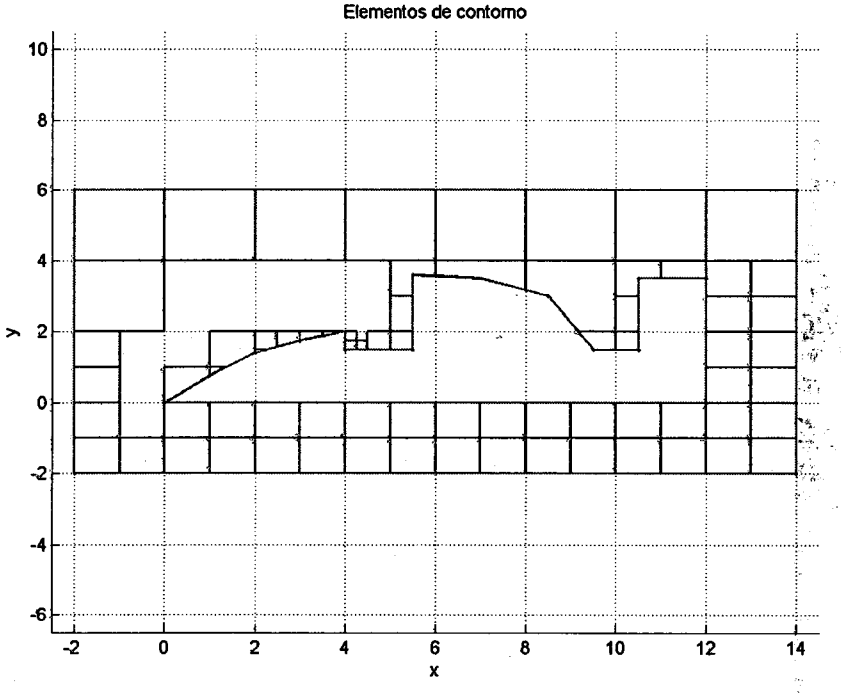
com todas as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.

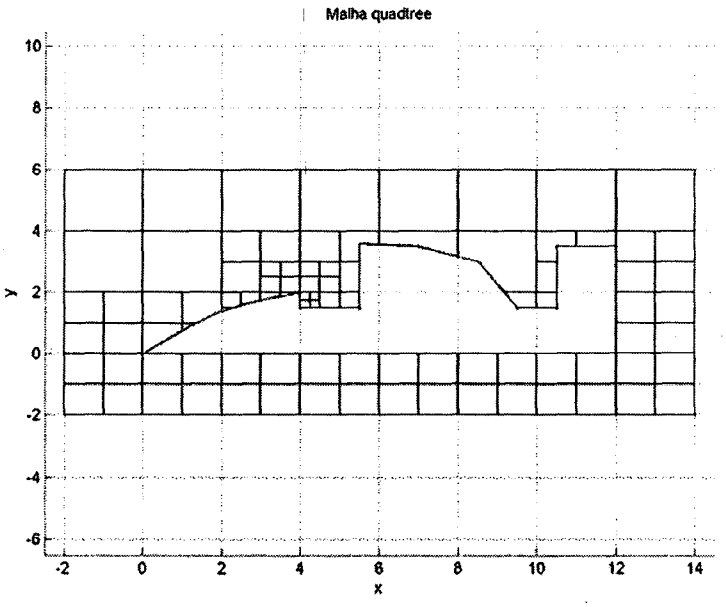



Quadtree restante
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x

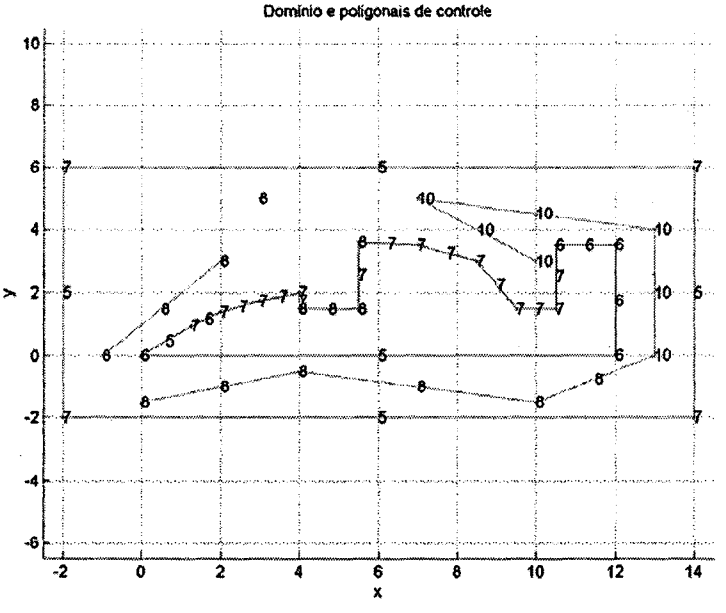

Quadtree básica por arestas
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x


Quadtree balanceada
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x

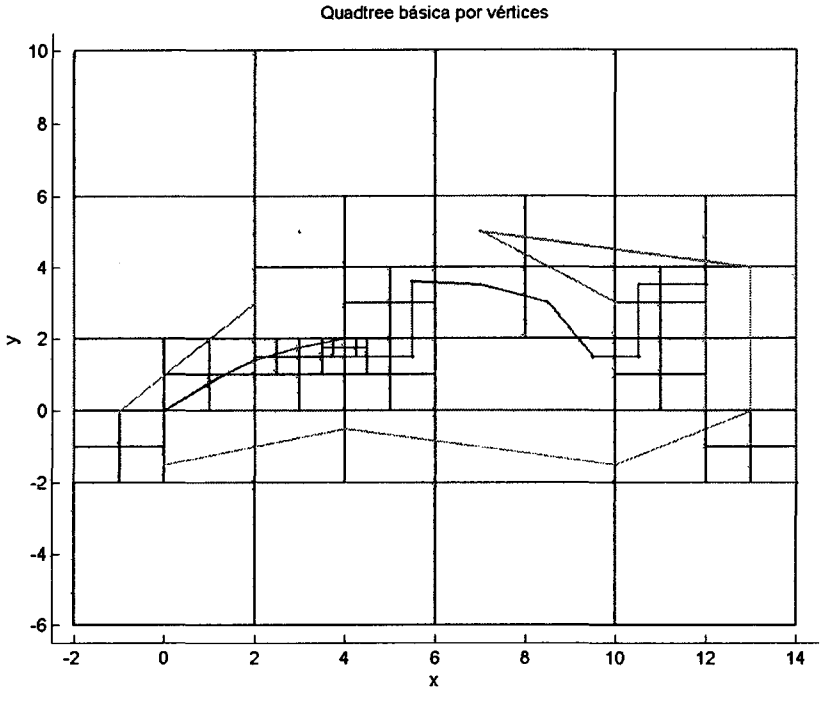

Elementos de contorno
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x

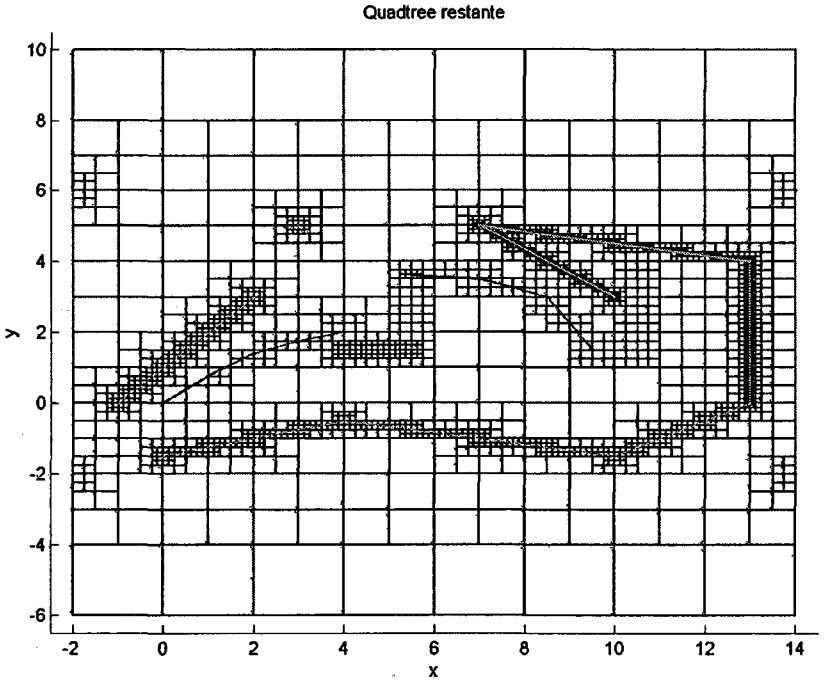


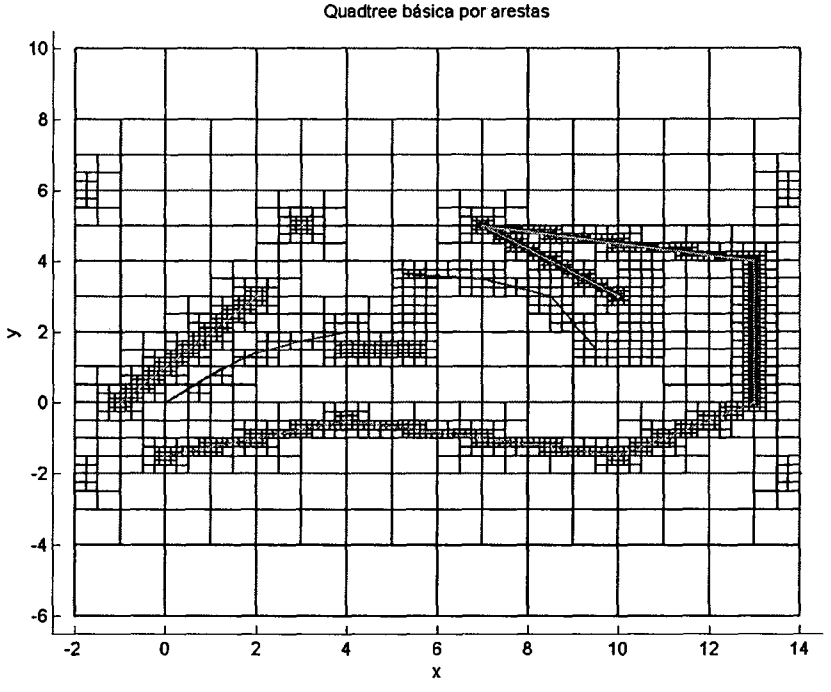
Finalmente, no problema 10, utilizaremos pela última vez este contorno base, adicionando algumas poligonais de controle ao domínio do problema 9, isto é,

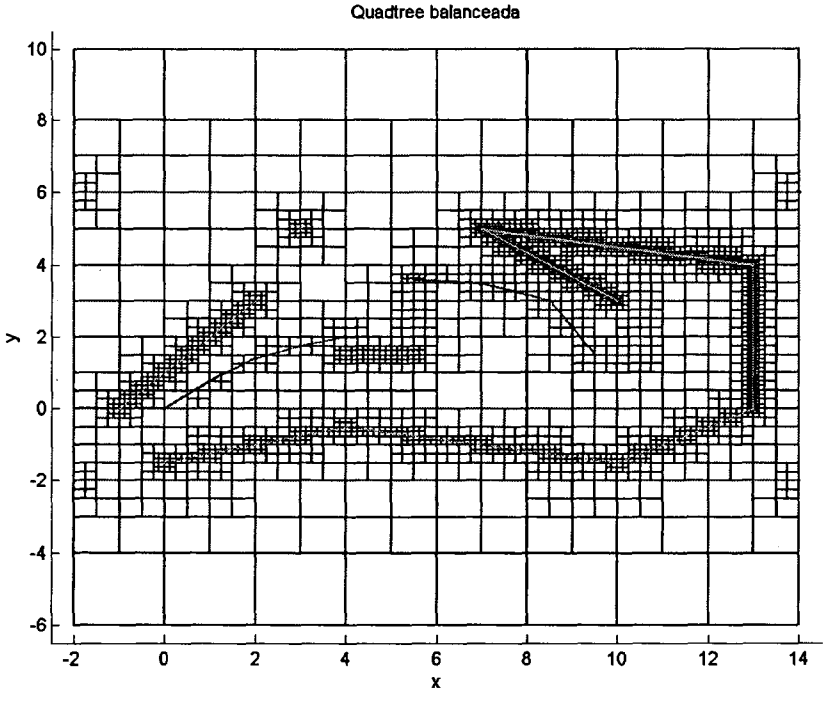


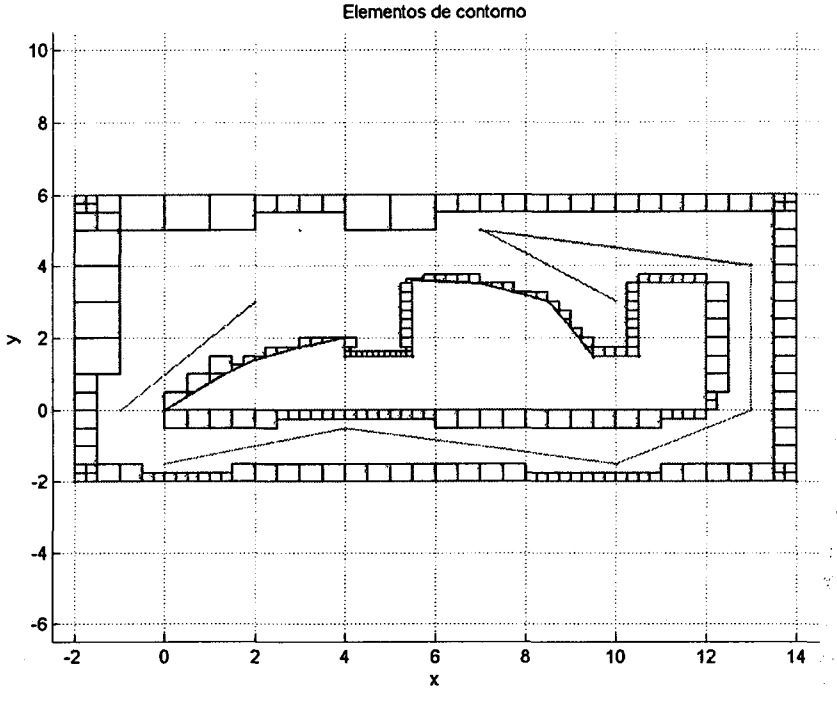
com todas as as etapas de geração da malha deste problema sendo ilustradas, sequencialmente, no que segue.

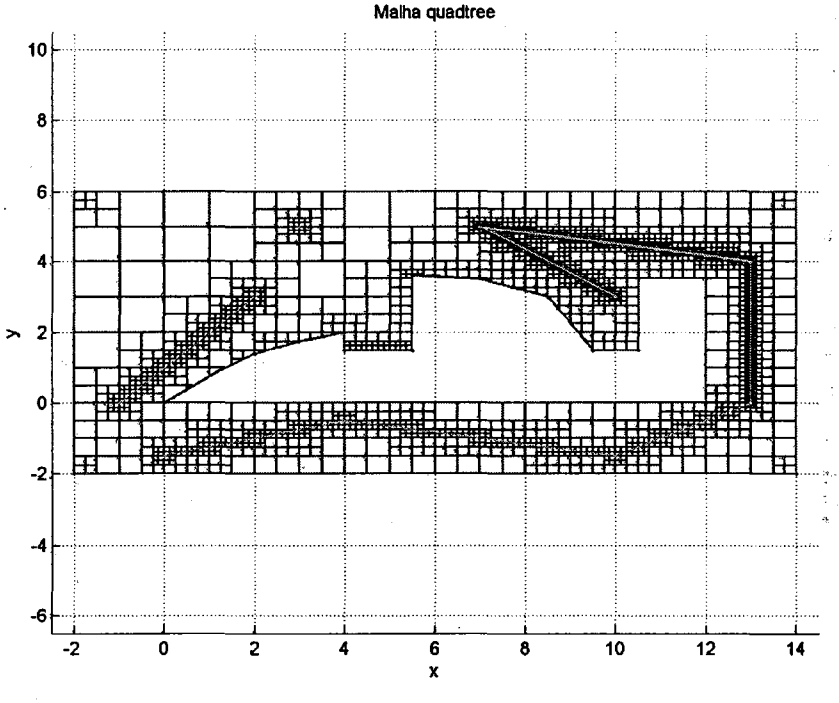

Quadtree básica por vértices
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x


Quadtree restante
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x


Quadtree básica por arestas
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x


Quadtree balanceada
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x


Elementos de contorno
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x


Malha quadtree
10
8
6
4
2
0
-2
-4
-6
y
-2
0
2
4
6
8
10
12
14
x

Capítulo 4

Conclusões

Por trás da elegante estrutura de dados associada à quadtree, ressaltamos duas vertentes presentes na implementação desta técnica de geração de malhas.

Na primeira delas, a natureza recursiva da estrutura de dados central (que é a árvore quaternária) se reflete no emprego de procedimentos também recursivos em cada uma das principais etapas do algoritmo de criação da malha. Para esta aplicação, especificamente, a utilização de recursividade contraria o senso-comum de muitos casos patológicos, nos quais procedimentos recursivos se mostram extremamente ineficientes mesmo que para tarefas simples [TB90]. Ainda no que diz respeito à utilização da recursividade na implementação desta técnica de geração de malhas, ela também contribui para uma melhor clareza do código ao propor atividades da forma "faça para a raiz da árvore, tome cada um dos filhos e faça a mesma coisa caso ele exista", as quais facilitam a implementação na medida que precisamos nos preocupar apenas no que fazer com a "raiz" da árvore, cabendo à recursão propagar essa ação aos nós restantes. Além disso, o caminhamento em árvore (que é empregado em todas as etapas da implementação da técnica) e a busca de alguma informação na mesma (como a procura pelos quadrantes que compõem o box de intersecção de um ponto de fronteira), também se tornam tarefas bastante "sintéticas" com o auxílio da recursão.

Em oposição, já na segunda vertente, existem diversos momentos da implementação que se faz necessária a realização de certos, digamos, "trabalhos sujos", os quais constituem a parte crítica do código, tanto pelo número de vezes que tais tarefas precisam ser executadas quanto pelo custo a elas associado (o qual é dito ser "alto" apenas se comparado às demais atividades realizadas).

Com o termo "trabalho sujo" nos referimos a todo teste de interceptação (por exemplo, verificar se dois segmentos se interceptam ou se um ponto pertence a um dado segmento ou certo quadrante) ou de posição relativa (por exemplo, em diversos momentos da implementação faz-se necessário verificar se um dado ponto está acima, abaixo ou sobre uma dada reta) que, eventualmente, precisa ser realizado, a fim de servir como chave para a tomada de alguma decisão específica. Tais testes exigem muita criatividade na manipulação de conceitos simples de Geometria Analítica, bem como cuidados adicionais no

tratamento de algumas comparações entre números reais[1].

Assim, podemos dizer que, embora a técnica em si de geração de uma malha quadtree seja relativamente simples, o "trabalho sujo" que precisa ser realizado é o momento crítico do processo. Apesar disso, afirmamos que tais dificuldades são irrisórias e perfeitamente tratáveis, proporcionando um meio bastante eficiente e robusto para a geração de malhas.

Uma propriedade bastante interessante da técnica quadtree diz respeito a sua flexibilidade quanto a realização de refinamentos. Parâmetros de refinamento podem ser especificados em qualquer poligonal (seja ela de fronteira ou não) contida no quadrante universo, exigindo um nível mínimo de subdivisões na região próxima a ela. Perceba também que a construção da quadtree mínima implica um nível mínimo de subdivisões (refinamento) para porções "delicadas" do domínio, o que mostra certa sensibilidade da técnica ao tratamento de porções específicas que exigiriam um nível mínimo de refinamento maior que o especificado como dado de entrada.

Além disso, pela própria natureza da técnica, altos índices de refinamento tendem a se concentrar cada vez mais em torno da poligonal que exige o refinamento, pois, a cada sudivisão de um quadrante, os quadrantes filhos que não mais interceptam tal poligonal deixam de ser novamente subdivididos.

Com isso, o refinamento pode ser facilmente concentrado em porções específicas, não influindo muito além dessa região. Se por um lado isto é bom, por outro deve existir um mecanismo adicional que proporcione uma melhor distribuição do refinamento, tendo em vista a suavidade na transição de elementos pequenos para elementos grandes.

Este controle de suavidade é realizado na técnica quadtree através do mecanismo de balanceamento local, sendo um meio que podemos classificar como razoável para o controle da razão de aspectos entre elementos da malha pela tolerância de desníveis. Isso porque, por sua definição, tal estratégia não afeta quadrantes terminais que façam fronteira "de canto", o que pode induzir a uma não suavidade na transição de elementos de tamanhos distintos em diagonal. Este, certamente, é um aspecto desta técnica de geração de malhas que pode e deve ser aprimorado.

Já outra característica extremamente boa da técnica quadtree é a sua ortogonalidade no interior da malha, em que se pese a presença de *hanging nodes*, que são os vértices de canto de um dado quadrante terminal que se localizam no meio de uma aresta de um quadrante vizinho, conforme ilustra a Figura 4.0.1. Embora o balanceamento local da quadtree amenize a presença desse fenômeno, ele só é totalmente eliminado quando $t = 0$, isto é, quando a malha é uniforme.

Ainda no que diz respeito à ortogonalidade, podemos verificar facilmente que a técnica quadtree não necessariamente gera malhas ortogonais aos contornos do domínio, sendo esta uma consequência direta do processo de recorte dos quadrantes que origina os elementos de fronteira.

Aliás, esse processo de recorte também é responsável por originar elementos de diver-

[1]Erros de arredondamento inerentes à representação finita de um número real podem fazer com que valores exatamente iguais sejam enxergados como diferentes pela máquina. Assim, em alguns casos, precisa-se estabelecer uma "tolerância" para comparações, tentando eliminar eventuais problemas com arredondamentos [QT00].

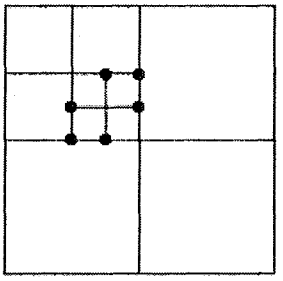


Figura 4.0.1: Os pontos vermelhos são exemplos de *hanging nodes* presentes em uma malha quadtree, com $t = 2$, gerada sobre um quadrado.

sas naturezas no contorno da malha. Tais elementos podem ser côncavos ou convexos, variando desde triângulos até heptágonos. Exigindo maiores níveis de refinamento, tal característica pode ser até amenizada, mas, dependendo do domínio considerado, nem sempre se faz possível eliminar esta ampla classe de elementos que assumem distintos formatos. Uma possível maneira de, talvez, resolver este problema seria utilizando uma triangulação que aproveitasse a estrutura da quadtree e de seus elementos, conforme aqui foram definidos.

Conforme foi dito no início do trabalho, consideramos aqui uma aresta dada por um segmento de reta. Obviamente, os algoritmos que foram apresentados possibilitam a extensão e utilização de arestas quaisquer, desde que satisfaçam à definição de aresta inicialmente apresentada.

Assim, uma outra possível melhoria da técnica aqui mostrada e implementada seria o tratamento de fronteiras curvas no domínio. A vantagem disto reside no fato que curvas não precisariam ser linearizadas, com a fronteira da malha sempre coincidindo com a fronteira do domínio (repare que, nos exemplos apresentados, as curvas tiveram que ser retificadas). Por outro lado, o inconveniente desta generalização seria a resolução de uma quantidade expressiva de sistemas que, se tornariam, possivelmente, não lineares (dependerá da parametrização da fronteira), para encontrar interceptações das fronteiras curvas com os quadrantes.

Finalmente, concluímos dizendo que a técnica quadtree proporciona algumas características interessantes para uma malha, bem como apresenta "brechas" que podem ser usadas para a extensão e melhoria da técnica, as quais devem ser investigadas, proporcionando um possível trabalho futuro.

Capítulo 5

O *QuadMesh2D*

5.1 Introdução

Conforme já foi amplamente citado no transcorrer do texto, as idéias inerentes à geração de uma malha quadtree, aqui apresentadas, foram implementadas em linguagem C, originando o *QuadMesh2D*[1].

Tendo em vista que o motivo de confecção do mesmo, neste trabalho, foi apenas a geração da malha em si, sua saída objetiva apenas este fim. Obviamente, dependendo do destino a ser dado a uma malha quadtree, pode-se modificar facilmente o módulo de saída de dados do *QuadMesh2D* a fim de que as informações relevantes da malha à aplicação possam ser convenientemente recuperadas via o salvamento em arquivo.

Assim, aqui, o processo de ilustração da malha, totalmente isento de qualquer emprego prático, se dá através de arquivos, escritos em linguagem M, e que podem ser executados tanto em MATLAB quanto no GNU Octave[2].

No que segue, apresentamos os códigos que compõem tal pacote, os quais foram empregados na geração dos resultados anteriormente apresentados. Salientamos que vastas explicações associadas a cada passagem do programa como um todo estão sob a forma de comentários junto ao código fonte, as quais julgamos ser suficientes para a compreensão de seu modo de operação.

5.2 Código fonte

No que segue, listamos todos os arquivos que compõem o QuadMesh2D, juntamente com as rotinas de representação gráfica da malha gerada.

[1] A implementação ocorreu por meio do programa "Dev C++ 4.9.9.2", disponível em http://www.bloodshed.net (acessado pela última vez em 6 de dezembro de 2007).

[2] Modificações nos m-scripts de representação podem ser exigidas dependendo da versão empregada dos aplicativos citados.

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : arqmanip.h
*****************************************************************************/

FILE *AbreArquivo(cadeia ARQF, cadeia MODO);
void FechaArquivo(FILE *fp);
void ListasIn(Vertice **ce, int *N, Head **ci, int *I, Head **pc, int *P);

#include "arqmanip.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : balanc.h
*****************************************************************************/

void InOrdem01(No* q);
void InOrdem23(No* q);
void InOrdem13(No* q);
void InOrdem02(No* q);
int MaxNivVizD(No *raiz, No*quad);
int MaxNivVizE(No *raiz, No*quad);
int MaxNivVizI(No *raiz, No*quad);
int MaxNivVizS(No *raiz, No*quad);
void ChecaBalanceamento(No *raiz, No* quad);

#include "balanc.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : base.h
*****************************************************************************/

/*****************************************************************************
 Parâmetros para a execução do código
*****************************************************************************/
#define epsm   1.0e-15         // tolerância nas comparações: épsilon da máquina
#define TOLER  1  // tolerância para o balanceamento : maior desnível permitido
                  // entre dois quadrantes vizinhos
#define ARQFCC "exemplo0.txt"  // arquivo físico com a entrada do programa
#define ARQFQA "quadexA0.txt"  // arquivo com a quadtree básica por vértices
#define ARQFQB "quadexB0.txt"  // arquivo com a quadtree restante
#define ARQFQC "quadexC0.txt"  // arquivo com a quadtree básica por arestas
#define ARQFQD "quadexD0.txt"  // arquivo com a quadtree balanceada
#define ARQFQE "quadexE0.txt"  // arquivo com os contornos discretizados
#define ARQFQF "quadexF0.txt"  // arquivo com a malha quadtree
#define MODOw  "w+"            // modo de acesso a arquivo físico para escrita
#define MODOr  "r"             // modo de acesso a arquivo físico para leitura
/*****************************************************************************/

#include "base.c"

No* AlocaNoQ();
void AlocaFilhos(No* raiz);
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : discont.h
*****************************************************************************/

void DefElemContorno(No *raiz);

#include "discont.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : extint.h
*****************************************************************************/

void CassaPosRel(No *raiz, double xr, double yr, Vertice *ce, int N, Head *ci,
                                                                    int I);

#include "extint.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : interceptos.h
*****************************************************************************/

char intercepta(No *raiz, double xc, double yc, double xt, double yt);
void detbox(No* raiz, double x, double y);
No* PtoInterceptaBox(double xc, double yc, double xt, double yt, double *xaux,
                                                                 double *yaux);

#include "interceptos.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : quadbasica.h
*****************************************************************************/

void CQBV(No *raiz, double x, double y, Vertice *ce, int N, Head *ci, int I);
void CQB(No *raiz, double xc, double yc, double xt, double yt, Vertice *ce,
                                                    int N, Head *ci, int I);

#include "quadbasica.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : restante.h
*****************************************************************************/

void CQR(No *raiz, double xc, double yc, double xt, double yt, int peso);

#include "restante.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : savedge.h
*****************************************************************************/

void AUXPercursoImprimeQ(No *raiz, FILE *entrada);
void PercursoImprimeQ(cadeia ARQFQ, No* raiz);
void AUXPercursoImprimeCDQ(No *raiz, FILE *entrada);
void PercursoImprimeCDQ(cadeia ARQFQ, No* raiz);
void AUXPercursoImprimeMQ(No *raiz, FILE *entrada);
void PercursoImprimeMQ(cadeia ARQFQ, No* raiz);

#include "savedge.c"
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : arqmanip.c
 Descrição : funções e procedimentos para a manipulação de arquivos
*****************************************************************************/

/*****************************************************************************
 Início A - funções que automatizam o processo de abertura e fechmento de
            arquivos físico/lógico
*****************************************************************************/
FILE *AbreArquivo(cadeia ARQF, cadeia MODO) {
     FILE *fp;
     fp = fopen(ARQF, MODO);
     if (fp == NULL) {
        printf("\n\nProblema no acesso ao arquivo \"%s\" no modo \"%s\"",
                                                              ARQF, MODO);
        getchar();
        exit(1);
     }
     return (fp);
}

void FechaArquivo(FILE *fp) {
     fclose(fp);
     return;
}
/***************************************************************** Fim A*/

/*****************************************************************************
 Início B - função que faz a leitura do arquivo que define os contornos
            (internos e externos) e poligonais de controle, construindo as
            listas: ce -> contorno externo
                    ci -> lista de cabeça das listas de contorno interno
                    pc -> lista de cabeça das listas de poligonais de controle
 OBS.: A entrada dos contornos do domínio e de controle ocorre por meio do
       arquivo físico ARQFCC. Este arquivo deve estar padronizado da seguinte
       forma, a menos dos comentários aqui efetuados:

 N          // -> número de vértices no ciclo externo que define o contorno
 x{i} y{i} pv{i} pa{i}  // -> segue uma sequência de i = 0,1,2,...,(N-1) linhas
                        //    definindo os atributos de cada vértice, onde:
                        //    x  - abcissa  do vértice
                        //    y  - ordenada do vértice
                        //    pv - nível mínimo que os quadrantes dos quais este
                        //         vértice faz parte devem apresentar
                        //    pa - nível mínimo que os quadrantes que envolvem a
                        //         aresta com origem nesse vértice devem possuir
```

```
                        // -> para simplificar algumas etapas, o programa assume
                        //    automaticamente o vértice indiciado por N igual
                        //    ao primeiro vértice passado, o que fecha o ciclo
                        //    fornecido e dá sentido à variável pa passada
                        //    juntamente com o último vértice como sendo o nível
                        //    exigido pela última aresta, que fecha o ciclo
 I+1        // -> número de ciclos internos presentes no domínio
 D[i]                   // -> segue uma sequência de i = 0,1,...,I blocos,
 x{j} y{j} pv{j} pa{j}  //    onde, para cada bloco, j = 0,1,2,...,(D[i]-1),
                        //    com o vértice de índice D[i] de cada ciclo sendo
                        //    definindo automaticamente pelo programa como
                        //    sendo igual ao primeiro vértice passado, o que
                        //    fecha o ciclo fornecido e dá sentido à variável
                        //    pa passada juntamente com o último vértice como
                        //    sendo o nível exigido pela última aresta, que
                        //    fecha o ciclo
 P+1        // -> número de poligonais de controle presentes no domínio
 C[i]                   // -> segue uma sequência de i = 0,1,...,P blocos,
 x{j} y{j} pv{j} pa{j}  //    onde, para cada bloco, j = 0,1,2,...,C[i].
                        // -> nas poligonais de controle cabe ao usuário
                        //    definir se elas serão abertas ou fechadas
                        // -> chamamos a atenção para o fato da definição de um
                        //    único ponto de controle ser equivalente à
                        //    definição de uma poligonal de comprimento zero,
                        //    isto é, para a qual temos apenas j=0, indicando
                        //    que ela é composta por um único vértice

 Ressaltamos que a definição do domínio se dá por um ciclo externo e uma coleção
 de ciclos internos, todos disjuntos. Para que o programa possa saber em qual
 região a malha deve ser gerada, a sequência de vértices que define o ciclo
 externo deve ser passada em uma orientação anti-horária, ao passo que, para
 ciclos internos, devemos assumir uma orientação horária para listar seus
 vértices no arquivo de entrada. Já as listas de controle, como não definem se
 uma região é externa ou interna ao domínio, podem possuir qualquer orientação
 (horária ou anti-horária) para a listagem de sua poligonal.
*****************************************************************************/
void ListasIn(Vertice **ce, int *N, Head **ci, int *I, Head **pc, int *P) {
     FILE *entrada;
     int ii, jj;
     entrada = AbreArquivo(ARQFCC,MODOr);

     fscanf(entrada,"%i\n",N);
     (*ce) = (Vertice *)malloc(((*N)+1)*sizeof(Vertice));
     for (ii = 0; ii < (*N); ii++) {
         fscanf(entrada,"%lf %lf %i %i\n",&((*ce)[ii].x), &((*ce)[ii].y),
                                          &((*ce)[ii].pv), &((*ce)[ii].pa));
     }
     (*ce)[(*N)].x  = (*ce)[0].x;
     (*ce)[(*N)].y  = (*ce)[0].y;
     (*ce)[(*N)].pv = (*ce)[0].pv;
     (*ce)[(*N)].pa = (*ce)[0].pa;  // este dado nunca será usado

     fscanf(entrada,"%i\n",I);
     (*ci) = (Head *)malloc((*I)*sizeof(Head));
     (*I)--;
```

```
    for (ii = 0; ii <= (*I); ii++) {
        fscanf(entrada,"%i\n", &((*ci)[ii].D));
        (*ci)[ii].lista = (Vertice *)malloc(((*ci)[ii].D + 1)*sizeof(Vertice));
        for (jj = 0; jj < (*ci)[ii].D; jj++) {
            fscanf(entrada,"%lf %lf %i %i\n", &((*ci)[ii].lista[jj].x),
                        &((*ci)[ii].lista[jj].y), &((*ci)[ii].lista[jj].pv),
                                                &((*ci)[ii].lista[jj].pa));
        }
        (*ci)[ii].lista[(*ci)[ii].D].x  = (*ci)[ii].lista[0].x;
        (*ci)[ii].lista[(*ci)[ii].D].y  = (*ci)[ii].lista[0].y;
        (*ci)[ii].lista[(*ci)[ii].D].pv = (*ci)[ii].lista[0].pv;
        (*ci)[ii].lista[(*ci)[ii].D].pa = (*ci)[ii].lista[0].pa;
        // este último dado nunca será usado
    }

    fscanf(entrada,"%i\n",P);
    (*pc) = (Head *)malloc((*P)*sizeof(Head));
    (*P)--;
    for (ii = 0; ii <= (*P); ii++) {
        fscanf(entrada,"%i\n", &((*pc)[ii].D));
        (*pc)[ii].lista = (Vertice *)malloc(((*pc)[ii].D)*sizeof(Vertice));
        (*pc)[ii].D--;
        for (jj = 0; jj <= (*pc)[ii].D; jj++) {
            fscanf(entrada,"%lf %lf %i %i\n", &((*pc)[ii].lista[jj].x),
                        &((*pc)[ii].lista[jj].y), &((*pc)[ii].lista[jj].pv),
                                                &((*pc)[ii].lista[jj].pa));
        }
    }

    FechaArquivo(entrada);
    return;
}
/************************************************************************** Fim B */
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : balanc.c
 Descrição : coleção de procedimentos para o balanceamento da quadtree
*****************************************************************************/

/*****************************************************************************
 Início A - conjunto de funções que percorrem uma sub-quadtee por um subconjunto
            de seus galhos em "in-ordem"
*****************************************************************************/
// função que percorre apenas os galhos formados pelos filhos 0 e 1
void InOrdem01(No* q) {
     if (q != NULL) {
        InOrdem01(q->f0);
        MN = (q->niv > MN) ? q->niv : MN;
        InOrdem01(q->f1);
     }
     return;
}

// função que percorre apenas os galhos formados pelos filhos 2 e 3
void InOrdem23(No* q) {
     if (q != NULL) {
        InOrdem23(q->f2);
        MN = (q->niv > MN) ? q->niv : MN;
        InOrdem23(q->f3);
     }
     return;
}

// função que percorre apenas os galhos formados pelos filhos 1 e 3
void InOrdem13(No* q) {
     if (q != NULL) {
        InOrdem13(q->f1);
        MN = (q->niv > MN) ? q->niv : MN;
        InOrdem13(q->f3);
     }
     return;
}

// função que percorre apenas os galhos formados pelos filhos 0 e 2
void InOrdem02(No* q) {
     if (q != NULL) {
        InOrdem02(q->f0);
        MN = (q->niv > MN) ? q->niv : MN;
        InOrdem02(q->f2);
     }
```

```
    return;
}
/************************************************************************ Fim A*/

/*****************************************************************************
 Início B - funções que determinam o máximo nível atingido pelos vizinhos
            laterais de um dado quadrante
*****************************************************************************/
// função que encontra o maior nível atingido por um quadrante vizinho a DIREITA
// de um determinado quadrante passado como parâmetro
int MaxNivVizD(No *raiz, No*quad) {
    char *rota;
    rota = (char *)malloc(((raiz->niv) - 1)*sizeof(char));
    int npasso = -1, // "npasso" e "rota" atuam como uma pilha
        passo = -1;
    No *q = raiz;
    while (passo != 0 && passo != 1 && q != quad) {
          if (q == ((q->pai)->f0)) {
             passo = 0;
          }
          else {
               if (q == ((q->pai)->f1)) {
                  passo = 1;
               }
               else {
                    if (q == ((q->pai)->f2)) {
                       passo = 2;
                    }
                    else {
                         if (q == ((q->pai)->f3)) {
                            passo = 3;
                         }
                    }
               }
          }
          npasso++;
          rota[npasso] = passo;
          q = q->pai;
    }
    // ... enquanto a pilha não esvaziar ou q não for folha (terminal)
    while ((npasso >= 0) && (q->f0 != NULL)) {
          passo = rota[npasso];
          switch (passo) {
                 case (0): q = q->f2; break;
                 case (1): q = q->f3; break;
                 case (2): q = q->f0; break;
                 case (3): q = q->f1; break;
          }
          npasso--;
    }
    free(rota);
    // neste ponto já se sabe que q é um ponteiro para um quadrante vizinho ao
    // quadrante apontado por raiz, bastando verificar qual o máximo nível
    // atingido pela subárvore de raiz q e "galhos" formados apenas pelos filhos
    // 0 e 1 (pois queremos os vizinhos a direita)
```

```
    MN = -1; // variável externa que retorna o máximo nível
    InOrdem01(q);
    return(MN);
}

// função que encontra o maior nível atingido por um quadrante vizinho ESQUERDO
// de um determinado quadrante passado como parâmetro
int MaxNivVizE(No *raiz, No*quad) {
    char *rota;
    rota = (char *)malloc(((raiz->niv) - 1)*sizeof(char));
    int npasso = -1, // "npasso" e "rota" atuam como uma pilha
        passo = -1;
    No *q = raiz;
    while (passo != 2 && passo != 3 && q != quad) {
          if (q == ((q->pai)->f0)) {
             passo = 0;
          }
          else {
               if (q == ((q->pai)->f1)) {
                  passo = 1;
               }
               else {
                    if (q == ((q->pai)->f2)) {
                       passo = 2;
                    }
                    else {
                         if (q == ((q->pai)->f3)) {
                            passo = 3;
                         }
                    }
               }
          }
          npasso++;
          rota[npasso] = passo;
          q = q->pai;
    }
    // ... enquanto a pilha não esvaziar ou q não for folha (terminal)
    while ((npasso >= 0) && (q->f0 != NULL)) {
          passo = rota[npasso];
          switch (passo) {
                 case (0): q = q->f2; break;
                 case (1): q = q->f3; break;
                 case (2): q = q->f0; break;
                 case (3): q = q->f1; break;
          }
          npasso--;
    }
    free(rota);
    // neste ponto já se sabe que q é um ponteiro para um quadrante vizinho ao
    // quadrante apontado por raiz, bastando verificar qual o máximo nível
    // atingido pela subárvore de raiz q e "galhos" formados apenas pelos filhos
    // 2 e 3 (pois queremos os vizinhos a esquerda)
    MN = -1; // variável externa que retorna o máximo nível
    InOrdem23(q);
    return(MN);
```

```
}

// função que encontra o maior nível atingido por um quadrante vizinho INFERIOR
// de um determinado quadrante passado como parâmetro
int MaxNivVizI(No *raiz, No*quad) {
    char *rota;
    rota = (char *)malloc(((raiz->niv) - 1)*sizeof(char));
    int npasso = -1, // "npasso" e "rota" atuam como uma pilha
        passo = -1;
    No *q = raiz;
    while (passo != 1 && passo != 3 && q != quad) {
          if (q == ((q->pai)->f0)) {
             passo = 0;
          }
          else {
               if (q == ((q->pai)->f1)) {
                  passo = 1;
               }
               else {
                    if (q == ((q->pai)->f2)) {
                       passo = 2;
                    }
                    else {
                         if (q == ((q->pai)->f3)) {
                            passo = 3;
                         }
                    }
               }
          }
          npasso++;
          rota[npasso] = passo;
          q = q->pai;
    }
    // ... enquanto a pilha não esvaziar ou q não for folha (terminal)
    while ((npasso >= 0) && (q->f0 != NULL)) {
          passo = rota[npasso];
          switch (passo) {
                 case (0): q = q->f1; break;
                 case (1): q = q->f0; break;
                 case (2): q = q->f3; break;
                 case (3): q = q->f2; break;
          }
          npasso--;
    }
    free(rota);
    // neste ponto já se sabe que q é um ponteiro para um quadrante vizinho ao
    // quadrante apontado por raiz, bastando verificar qual o máximo nível
    // atingido pela subárvore de raiz q e "galhos" formados apenas pelos filhos
    // 1 e 3 (pois queremos os vizinhos a inferiores)
    MN = -1; // variável externa que retorna o máximo nível
    InOrdem13(q);
    return(MN);
}

// função que encontra o maior nível atingido por um quadrante vizinho SUPERIOR
```

```
// de um determinado quadrante passado como parâmetro
int MaxNivVizS(No *raiz, No*quad) {
    char *rota;
    rota = (char *)malloc(((raiz->niv) - 1)*sizeof(char));
    int npasso = -1, // "npasso" e "rota" atuam como uma pilha
        passo = -1;
    No *q = raiz;
    while (passo != 0 && passo != 2 && q != quad) {
          if (q == ((q->pai)->f0)) {
             passo = 0;
          }
          else {
               if (q == ((q->pai)->f1)) {
                  passo = 1;
               }
               else {
                    if (q == ((q->pai)->f2)) {
                       passo = 2;
                    }
                    else {
                         if (q == ((q->pai)->f3)) {
                            passo = 3;
                         }
                    }
               }
          }
          npasso++;
          rota[npasso] = passo;
          q = q->pai;
    }
    // ... enquanto a pilha não esvaziar ou q não for folha (terminal)
    while ((npasso >= 0) && (q->f0 != NULL)) {
          passo = rota[npasso];
          switch (passo) {
                 case (0): q = q->f1; break;
                 case (1): q = q->f0; break;
                 case (2): q = q->f3; break;
                 case (3): q = q->f2; break;
          }
          npasso--;
    }
    free(rota);
    // neste ponto já se sabe que q é um ponteiro para um quadrante vizinho ao
    // quadrante apontado por raiz, bastando verificar qual o máximo nível
    // atingido pela subárvore de raiz q e "galhos" formados apenas pelos filhos
    // 0 e 2 (pois queremos os vizinhos superiores).
    MN = -1; // variável externa que retorna o máximo nível
    InOrdem02(q);
    return(MN);
}
/************************************************************************ Fim B*/

/*****************************************************************************
 Início C - procedimento que percorre a quadtree em busca de desníveis mais
            intenos entre quadrantes vizinhos que a tolerância máxima permitida,
```

```
            fazendo uso inteso das fuções acima especificadas
*****************************************************************************/
void ChecaBalanceamento(No *raiz, No* quad) {
     if (raiz != NULL) {
        if (raiz->f0 == NULL) { // verifica se raiz é folha (quadrante terminal)
           int maxnv = -1,
               res = - 1;
           if (raiz->xmax < quad->xmax) { // testa se existe quadrante
                                          // a direita de raiz
              res = MaxNivVizD(raiz,quad);
              maxnv = (maxnv > res) ? maxnv : res;
           }
           if (raiz->xmin > quad->xmin) { // testa se existe quadrante
                                          // a esquerda de raiz
              res = MaxNivVizE(raiz,quad);
              maxnv = (maxnv > res) ? maxnv : res;
           }
           if (raiz->ymin > quad->ymin) { // testa se existe quadrante
                                          // abaixo de raiz
              res = MaxNivVizI(raiz,quad);
              maxnv = (maxnv > res) ? maxnv : res;
           }
           if (raiz->ymax < quad->ymax) { // testa se existe quadrante
                                          // acima de raiz
              res = MaxNivVizS(raiz,quad);
              maxnv = (maxnv > res) ? maxnv : res;
           }
           if (maxnv > raiz->niv + TOLER) {
              AlocaFilhos(raiz);
              flag = 1;
           }
        }
        ChecaBalanceamento(raiz->f0,quad);
        ChecaBalanceamento(raiz->f1,quad);
        ChecaBalanceamento(raiz->f2,quad);
        ChecaBalanceamento(raiz->f3,quad);
     }
     return;
}
/***************************************************************************** Fim C*/
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : base.c
 Descrição : definição de: -> tipos e estruturas de dados
                           -> manipulações básicas das estruturas de dados
                           -> variáveis globais
*****************************************************************************/

/*****************************************************************************
 Início A - tipos e estruturas de dados
*****************************************************************************/
// estrutura de dados para representar cada vértice do contorno
typedef struct {
        double x,  // abcissa  do vértice
               y;  // ordenada do vértice
        int pv,    // nível mínimo exigido por esse vértice na quadtree
            pa;    // nível mínimo exigido pela aresta iniciada neste vértice
} Vertice;

// cabeça da lista associada a cada ciclo interno ou poligonal de controle
typedef struct {
        int D;          // número de vértices presentes nesta lista
        Vertice *lista; // ponteiro para a lista de vértices que compõem este
                        // ciclo interno ou poligonal de controle
} Head;

// nó da quadtree
typedef struct no {
        struct no *pai,  // ponteiro para o pai do nó considerado
                  *f0,
                  *f1,   // fi, i = 0,1,2,3, representam ponteiros para cada
                  *f2,   // um dos filhos de cada nó
                  *f3;
        double *xelem,   // vetores com a abcissa e a ordenada de cada vértice
               *yelem;   // que irá compor um "ciclo" do quadrante que é
                         // recortado para formar um elemento finito de contorno
        int niv;  // indica o nível do nó na árvore (a raiz tem nível 1)
        char pos; // este campo é útil apenas para os quadrantes terminais
                  // pos = -3 -> quadrante externo ao domínio
                  //     = -2 -> quadrante interno ao domínio
                  //     = 0,1,2,3,4,5,6,7 -> quadrante de fronteira e o número
                  //       armazenado faz menção ao número de vértices do
                  //       elemento finito criado a partir desse quadrante,
                  //       dando o número de elementos dos vetores dos campos
                  //       xelem e yelem desta estrutura, indexados 0,1,...,pos.
        double xmin,  // variáveis que estabelecem o limite do quadrante fechado
               xmax,
```

```
                ymin,
                ymax;
} No;

// cadeia de caracteres
typedef char cadeia[30];
/************************************************************************** Fim A*/

/*****************************************************************************
 Início B - variáveis globais
*****************************************************************************/
char tam = -1;
No* box[4];     // lista com os quadrantes que compõem um dado box de intersecção
char flag = 0; // variável usada para comunicação entre diversas rotinas
int MN = 0;     // aramzenará o maior nível atingido por um quadrante vizinho
/************************************************************************** Fim B*/

/*****************************************************************************
 Início C - funções empregadas na alocação de quadrantes para a quadtree
*****************************************************************************/
// procedimento que automatiza a criação de um quadrante
No* AlocaNoQ() {
    No* q;
    q = (No*)malloc(sizeof(No));
    if (q == NULL) {
       printf("\nFalha ao alocar memoria -> execucao abortada.");
       getchar();
       getchar();
       exit(1);
    }
    q->pai   = NULL;
    q->f0    = NULL;
    q->f1    = NULL;
    q->f2    = NULL;
    q->f3    = NULL;
    q->xelem = NULL;
    q->yelem = NULL;
    q->pos   = -1;   // inicialmente, todos os quadrantes recebem no campo pos
    q->xmax  = 0;    // um valor inválido segundo a legenda apresentada na
    q->xmin  = 0;    // definição do tipo No: este valor -1 é estratégico para
    q->ymax  = 0;    // a criação da lista que forma o ciclo que define o
    q->ymin  = 0;    // contorno do elemento finito que intercepta alguma
    return(q);       // fronteira (interna ou externa) do domínio
}

// função para alocar os quatro filhos para uma folha (quadante terminal)
void AlocaFilhos(No* raiz) {
     No *aux;
     double xmedio,
            ymedio;
     xmedio = 0.5*(raiz->xmin + raiz->xmax);
     ymedio = 0.5*(raiz->ymin + raiz->ymax);
     aux = AlocaNoQ();        // quadrante 0
     aux->niv = raiz->niv + 1;
     aux->pai  = raiz;
```

```
    aux->xmin = raiz->xmin;
    aux->xmax = xmedio;
    aux->ymin = raiz->ymin;
    aux->ymax = ymedio;
    raiz->f0 = aux;
    aux = AlocaNoQ();          // quadrante 1
    aux->niv = raiz->niv + 1;
    aux->pai  = raiz;
    aux->xmin = raiz->xmin;
    aux->xmax = xmedio;
    aux->ymin = ymedio;
    aux->ymax = raiz->ymax;
    raiz->f1 = aux;
    aux = AlocaNoQ();          // quadrante 2
    aux->niv = raiz->niv + 1;
    aux->pai  = raiz;
    aux->xmin = xmedio;
    aux->xmax = raiz->xmax;
    aux->ymin = raiz->ymin;
    aux->ymax = ymedio;
    raiz->f2 = aux;
    aux = AlocaNoQ();          // quadrante 3
    aux->niv = raiz->niv + 1;
    aux->pai  = raiz;
    aux->xmin = xmedio;
    aux->xmax = raiz->xmax;
    aux->ymin = ymedio;
    aux->ymax = raiz->ymax;
    raiz->f3 = aux;
    return;
}
/************************************************************************** Fim C*/
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : discont.c
 Descrição : função que auxilia o processo de discretização dos contornos do
             domínio definido pelo usuário
*****************************************************************************/

/*****************************************************************************
 Início A - uma vez que o contorno já esteja discretizado, esta função cria os
            ciclos que definem os elementos de contorno
*****************************************************************************/
void DefElemContorno(No *raiz) {
     if (raiz != NULL) {
        if (raiz->f0 == NULL) { // testa se raiz é folha
           if (raiz->pos >= 0) { // testa se o quadrante é de contorno
              double x0, x1, y0, y1,
                     xa, xb, ya, yb,
                     xt, yt,
                     c;
              flag = 1;
              // enquanto o ciclo que define o elemento não for fechado...
              while (raiz->xelem[0] != raiz->xelem[raiz->pos] ||
                                       raiz->yelem[0] != raiz->yelem[raiz->pos] ) {
                    flag = 1; // o controle flag garante a adição de um único
                              // elemento à lista a cada laço
                    //aresta superior
                    if (flag == 1) {
                       //extremos da aresta
                       x0 = raiz->xmin;
                       x1 = raiz->xmax;
                       y0 = raiz->ymax;
                       y1 = raiz->ymax;
                       // último segmento presente no contorno do ciclo
                       xa = raiz->xelem[raiz->pos-1];
                       xb = raiz->xelem[raiz->pos];
                       ya = raiz->yelem[raiz->pos-1];
                       yb = raiz->yelem[raiz->pos];
                       // testa se o extremo da lista está na aresta superior
                       if (x0 - epsm <= xb && xb <= x1 + epsm &&
                                                    fabs(yb-y0) <= epsm) {
                          // testa se o ciclo pode ser fechado
                          xt = raiz->xelem[0];
                          yt = raiz->yelem[0];
                          c = (yt-ya)*(xb-xa) + (yb-ya)*(xa-xt);
                          if (x0 - epsm <= xt && xt <= x1 + epsm &&
                                                 fabs(yt-y0) <= epsm && c > 0) {
                             flag = 0;
```

```
            (raiz->pos)++;
            raiz->xelem[raiz->pos] = xt;
            raiz->yelem[raiz->pos] = yt;
         }
         // se o ciclo não foi fechado, testo os dois extremos
         // da aresta do quadrante
         else {
              c = (y0-ya)*(xb-xa) + (yb-ya)*(xa-x0);
              if (c > 0) {
                 flag = 0;
                 (raiz->pos)++;
                 raiz->xelem[raiz->pos] = x0;
                 raiz->yelem[raiz->pos] = y0;
              }
              else {
                   c = (y1-ya)*(xb-xa) + (yb-ya)*(xa-x1);
                   if (c > 0) {
                      flag = 0;
                      (raiz->pos)++;
                      raiz->xelem[raiz->pos] = x1;
                      raiz->yelem[raiz->pos] = y1;
                   }
              }
         }
      }
   }

   //aresta inferior
   if (flag == 1) {
      //extremos da aresta
      x0 = raiz->xmin;
      x1 = raiz->xmax;
      y0 = raiz->ymin;
      y1 = raiz->ymin;
      // último segmento presente no contorno do ciclo
      xa = raiz->xelem[raiz->pos-1];
      xb = raiz->xelem[raiz->pos];
      ya = raiz->yelem[raiz->pos-1];
      yb = raiz->yelem[raiz->pos];
      // testa se o extremo da lista está na aresta inferior
      if (x0 - epsm <= xb && xb <= x1 + epsm &&
                                          fabs(yb-y0) <= epsm) {
         // testa se o ciclo pode ser fechado
         xt = raiz->xelem[0];
         yt = raiz->yelem[0];
         c = (yt-ya)*(xb-xa) + (yb-ya)*(xa-xt);
         if (x0 - epsm <= xt && xt <= x1 + epsm &&
                                fabs(yt-y0) <= epsm && c > 0) {
            flag = 0;
            (raiz->pos)++;
            raiz->xelem[raiz->pos] = xt;
            raiz->yelem[raiz->pos] = yt;
         }
         // se o ciclo não foi fechado, testo os dois extremos
         // da aresta do quadrante
```

```
            else {
                 c = (y0-ya)*(xb-xa) + (yb-ya)*(xa-x0);
                 if (c > 0) {
                    flag = 0;
                    (raiz->pos)++;
                    raiz->xelem[raiz->pos] = x0;
                    raiz->yelem[raiz->pos] = y0;
                 }
                 else {
                      c = (y1-ya)*(xb-xa) + (yb-ya)*(xa-x1);
                      if (c > 0) {
                         flag = 0;
                         (raiz->pos)++;
                         raiz->xelem[raiz->pos] = x1;
                         raiz->yelem[raiz->pos] = y1;
                      }
                 }
            }
       }
  }

  //aresta direita
  if (flag == 1) {
     //extremos da aresta
     x0 = raiz->xmax;
     x1 = raiz->xmax;
     y0 = raiz->ymin;
     y1 = raiz->ymax;
     // último segmento presente no contorno do ciclo
     xa = raiz->xelem[raiz->pos-1];
     xb = raiz->xelem[raiz->pos];
     ya = raiz->yelem[raiz->pos-1];
     yb = raiz->yelem[raiz->pos];
     // testa se o extremo da lista está na aresta direita
     if (y0 - epsm <= yb && yb <= y1 + epsm &&
                                          fabs(xb-x0) <= epsm) {
        // testa se o ciclo pode ser fechado
        xt = raiz->xelem[0];
        yt = raiz->yelem[0];
        c = (yt-ya)*(xb-xa) + (yb-ya)*(xa-xt);
        if (y0 - epsm <= yt && yt <= y1 + epsm &&
                               fabs(xt-x0) <= epsm && c > 0) {
           flag = 0;
           (raiz->pos)++;
           raiz->xelem[raiz->pos] = xt;
           raiz->yelem[raiz->pos] = yt;
        }
        // se o ciclo não foi fechado, testo os dois extremos
        // da aresta do quadrante
        else {
             c = (y0-ya)*(xb-xa) + (yb-ya)*(xa-x0);
             if (c > 0) {
                flag = 0;
                (raiz->pos)++;
                raiz->xelem[raiz->pos] = x0;
```

```
                raiz->yelem[raiz->pos] = y0;
            }
            else {
                c = (y1-ya)*(xb-xa) + (yb-ya)*(xa-x1);
                if (c > 0) {
                    flag = 0;
                    (raiz->pos)++;
                    raiz->xelem[raiz->pos] = x1;
                    raiz->yelem[raiz->pos] = y1;
                }
            }
        }
    }
}

//aresta esquerda
if (flag == 1) {
    //extremos da aresta
    x0 = raiz->xmin;
    x1 = raiz->xmin;
    y0 = raiz->ymin;
    y1 = raiz->ymax;
    // último segmento presente no contorno do ciclo
    xa = raiz->xelem[raiz->pos-1];
    xb = raiz->xelem[raiz->pos];
    ya = raiz->yelem[raiz->pos-1];
    yb = raiz->yelem[raiz->pos];
    // testa se o extremo da lista está na aresta esquerda
    if (y0 - epsm <= yb && yb <= y1 + epsm &&
                                        fabs(xb-x0) <= epsm) {
        // testa se o ciclo pode ser fechado
        xt = raiz->xelem[0];
        yt = raiz->yelem[0];
        c = (yt-ya)*(xb-xa) + (yb-ya)*(xa-xt);
        if (y0 - epsm <= yt && yt <= y1 + epsm &&
                                fabs(xt-x0) <= epsm && c > 0) {
            flag = 0;
            (raiz->pos)++;
            raiz->xelem[raiz->pos] = xt;
            raiz->yelem[raiz->pos] = yt;
        }
        // se o ciclo não foi fechado, testo os dois extremos
        // da aresta do quadrante
        else {
            c = (y0-ya)*(xb-xa) + (yb-ya)*(xa-x0);
            if (c > 0) {
                flag = 0;
                (raiz->pos)++;
                raiz->xelem[raiz->pos] = x0;
                raiz->yelem[raiz->pos] = y0;
            }
            else {
                c = (y1-ya)*(xb-xa) + (yb-ya)*(xa-x1);
                if (c > 0) {
                    flag = 0;
```

```
                        (raiz->pos)++;
                        raiz->xelem[raiz->pos] = x1;
                        raiz->yelem[raiz->pos] = y1;
                    }
                }
            }
        }
    }
    // se neste ponto ainda tivermos flag = 1 quer dizer que o
    // segmento de contorno já discretizado na lista
    // (xelem,yelem) está estrita e inteiramente contido em uma
    // aresta do quadrante; neste caso, considerando os dois
    // pontos extremos do lado do quadrante, determino o que
    // preservará a orientação do ciclo para ser armazenado na
    // lista (xelem,yelem)
    if (flag == 1) {
        //aresta superior
        if (flag == 1) {
            //extremos da aresta
            x0 = raiz->xmin;
            x1 = raiz->xmax;
            y0 = raiz->ymax;
            y1 = raiz->ymax;
            // último segmento presente no contorno do ciclo
            xa = raiz->xelem[raiz->pos-1];
            xb = raiz->xelem[raiz->pos];
            ya = raiz->yelem[raiz->pos-1];
            yb = raiz->yelem[raiz->pos];
            // testa se o extremo da lista está na aresta superior
            if (x0 - epsm <= xb && xb <= x1 + epsm &&
                                            fabs(yb-y0) <= epsm) {
                if ((xb - xa >= 0 && x0 - xa >= xb - xa) ||
                            (xb - xa < 0 && x0 - xa <= xb - xa)) {
                    flag = 0;
                    (raiz->pos)++;
                    raiz->xelem[raiz->pos] = x0;
                    raiz->yelem[raiz->pos] = y0;

                }
                else {
                     flag = 0;
                     (raiz->pos)++;
                     raiz->xelem[raiz->pos] = x1;
                     raiz->yelem[raiz->pos] = y1;
                }
            }
        }

        //aresta inferior
        if (flag == 1) {
            //extremos da aresta
            x0 = raiz->xmin;
            x1 = raiz->xmax;
            y0 = raiz->ymin;
            y1 = raiz->ymin;
```

```
   // último segmento presente no contorno do ciclo
   xa = raiz->xelem[raiz->pos-1];
   xb = raiz->xelem[raiz->pos];
   ya = raiz->yelem[raiz->pos-1];
   yb = raiz->yelem[raiz->pos];
   // testa se o extremo da lista está na aresta inferior
   if (x0 - epsm <= xb && xb <= x1 + epsm &&
                                    fabs(yb-y0) <= epsm) {
      if ((xb - xa >= 0 && x0 - xa >= xb - xa) ||
                  (xb - xa < 0 && x0 - xa <= xb - xa)) {
         flag = 0;
         (raiz->pos)++;
         raiz->xelem[raiz->pos] = x0;
         raiz->yelem[raiz->pos] = y0;

      }
      else {
           flag = 0;
           (raiz->pos)++;
           raiz->xelem[raiz->pos] = x1;
           raiz->yelem[raiz->pos] = y1;
      }
   }
}

//aresta direita
if (flag == 1) {
   //extremos da aresta
   x0 = raiz->xmax;
   x1 = raiz->xmax;
   y0 = raiz->ymin;
   y1 = raiz->ymax;
   // último segmento presente no contorno do ciclo
   xa = raiz->xelem[raiz->pos-1];
   xb = raiz->xelem[raiz->pos];
   ya = raiz->yelem[raiz->pos-1];
   yb = raiz->yelem[raiz->pos];
   // testa se o extremo da lista está na aresta direita
   if (y0 - epsm <= yb && yb <= y1 + epsm &&
                                    fabs(xb-x0) <= epsm) {
      if ((yb - ya >= 0 && y0 - ya >= yb - ya) ||
                  (yb - ya < 0 && y0 - ya <= yb - ya)) {
         flag = 0;
         (raiz->pos)++;
         raiz->xelem[raiz->pos] = x0;
         raiz->yelem[raiz->pos] = y0;

      }
      else {
           flag = 0;
           (raiz->pos)++;
           raiz->xelem[raiz->pos] = x1;
           raiz->yelem[raiz->pos] = y1;
      }
   }
```

```
                    }

                    //aresta esquerda
                    if (flag == 1) {
                       //extremos da aresta
                       x0 = raiz->xmin;
                       x1 = raiz->xmin;
                       y0 = raiz->ymin;
                       y1 = raiz->ymax;
                       // último segmento presente no contorno do ciclo
                       xa = raiz->xelem[raiz->pos-1];
                       xb = raiz->xelem[raiz->pos];
                       ya = raiz->yelem[raiz->pos-1];
                       yb = raiz->yelem[raiz->pos];
                       // testa se o extremo da lista está na aresta esquerda
                       if (y0 - epsm <= yb && yb <= y1 + epsm &&
                                                    fabs(xb-x0) <= epsm) {
                          if ((yb - ya >= 0 && y0 - ya >= yb - ya) ||
                                       (yb - ya < 0 && y0 - ya <= yb - ya)) {
                             flag = 0;
                             (raiz->pos)++;
                             raiz->xelem[raiz->pos] = x0;
                             raiz->yelem[raiz->pos] = y0;

                          }
                          else {
                               flag = 0;
                               (raiz->pos)++;
                               raiz->xelem[raiz->pos] = x1;
                               raiz->yelem[raiz->pos] = y1;
                          }
                       }
                    }
                 }
              }
           }
        }
        DefElemContorno(raiz->f0);
        DefElemContorno(raiz->f1);
        DefElemContorno(raiz->f2);
        DefElemContorno(raiz->f3);
     }
     return;
}
/************************************************************************ Fim A */
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : extint.c
 Descrição : procedimento que define a posição relativa de cada quadrante fora
             do contorno, analisando se ele é interno ou externo ao domínio
*****************************************************************************/

/*****************************************************************************
 Início A - esta função varre a quadtree e checa, para os quadrantes que não são
            de contorno se eles devem ou não representar um elemento finito da
            malha (assim, verificamos quais são os quadrantes internos e quais
            são os externos)
OBS.: assumimos aqui que os quadrantes de contorno já foram determinados,
      restando apenas verificar quais são os externos ou não, o que pode ser
      conseguido testando se um ponto interno ao quadrante está dentro ou
      fora do domínio
*****************************************************************************/
void CassaPosRel(No *raiz, double xr, double yr, Vertice *ce, int N, Head *ci,
                                                                    int I) {
    if (raiz != NULL) {
       if (raiz->f0 == NULL) { // testa se raiz é folha
          if (raiz->pos < 0) { // testa se o elemento não é de contorno
             double x0, y0, x1, y1,
                    bx, by,  // baricentro do quadrante
                    csi, // parâmetro do segmento (bx,by)--(xr,yr)
                    eta, // parâmetro do segmento [(x0,y0)--(y1,y1)[
                    aux1, aux2, aux3, aux4,
                    det;
             int ii, jj,
                 rel;  // variável com a posição relativa do quadrante
             bx = (raiz->xmax + raiz->xmin)*0.5;
             by = (raiz->ymax + raiz->ymin)*0.5;

             rel = 0;
             // teste de interceptações com o ciclo externo
             for (ii = 0; ii < N; ii++) {
                 x0 = ce[ii].x;
                 y0 = ce[ii].y;
                 x1 = ce[ii+1].x;
                 y1 = ce[ii+1].y;
                 aux1 = (x0 - bx)/(xr-bx);
                 aux2 = (y0 - by)/(yr-by);
                 aux3 = (x1 - bx)/(xr-bx);
                 aux4 = (y1 - by)/(yr-by);
                 // teste para saber se o segmento[(x0,y0)--(y1,y1)[ está sobre
                 // o segmento (bx,by)--(xr,yr)
                 if (!(aux1 == aux2 && aux3 == aux4 && 0 <= aux1 && aux1 <=1 &&
```

```
                                                    0 <= aux3 && aux3 <=1)) {
                det = -((xr-bx)*(y1-y0) - (x1-x0)*(yr-by));
                // testa se segmento e aresta podem se interceptar
                if (det != 0) {
                   csi = (-(y1-y0)*(x0 - bx) + (x1-x0)*(y0 - by))/det;
                   eta = (-(yr-by)*(x0 - bx) + (xr-bx)*(y0 - by))/det;

                   if (0 <= csi && csi <= 1 && 0 <= eta && eta < 1) {
                      rel++;
                   }
                }
             }
          }
          // teste de interceptações com os ciclos internos
          for (ii = 0; ii <= I; ii++) {
             for (jj = 0; jj < ci[ii].D; jj++) {
                x0 = ci[ii].lista[jj].x;
                y0 = ci[ii].lista[jj].y;
                x1 = ci[ii].lista[jj+1].x;
                y1 = ci[ii].lista[jj+1].y;
                aux1 = (x0 - bx)/(xr-bx);
                aux2 = (y0 - by)/(yr-by);
                aux3 = (x1 - bx)/(xr-bx);
                aux4 = (y1 - by)/(yr-by);
                // teste para saber se o segmento[(x0,y0)--(y1,y1)[ está
                // sobre o segmento (bx,by)--(xr,yr)
                if (!(aux1 == aux2 && aux3 == aux4 && 0 <= aux1 && aux1 <=1
                                                   && 0 <= aux3 && aux3 <=1)) {
                   det = -((xr-bx)*(y1-y0) - (x1-x0)*(yr-by));
                   // testa se segmento e aresta podem se interceptar
                   if (det != 0) {
                      csi = (-(y1-y0)*(x0 - bx) + (x1-x0)*(y0 - by))/det;
                      eta = (-(yr-by)*(x0 - bx) + (xr-bx)*(y0 - by))/det;
                      // a aresta tem segunda extremidade aberta (eta < 1)
                      if (0 <= csi && csi <= 1 && 0 <= eta && eta < 1) {
                         rel++;
                      }
                   }
                }
             }
          }
          if (rel % 2 == 0) {
             raiz->pos = -3;
          }
          else {
               raiz->pos = -2;
          }
       }
    }
    CassaPosRel(raiz->f0,xr,yr,ce,N,ci,I);
    CassaPosRel(raiz->f1,xr,yr,ce,N,ci,I);
    CassaPosRel(raiz->f2,xr,yr,ce,N,ci,I);
    CassaPosRel(raiz->f3,xr,yr,ce,N,ci,I);
}
return;
```

```
}
/************************************************************************** Fim A*/
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : interceptos.c
 Descrição : este arquivo agrega funções que testam ou calculam intersecções
             relacionadas a segmentos e quadrantes
*****************************************************************************/

/*****************************************************************************
 Início A - função que aceita como parâmetros um quadrante e extremos de um
            segmento, retornando: 1 - se segmento intercepta quadrante
                                  0 - se segmento não intrercepta quadrante
*****************************************************************************/
char intercepta(No *raiz, double xc, double yc, double xt, double yt) {
     // caso em que o segmento é um vertice
     if (xc == xt && yc == yt) {
        if ((raiz->xmin <= xc && xc <= raiz->xmax) && (raiz->ymin <= yc &&
                                                       yc <= raiz->ymax)) {
           return(1);
        }
        else {
             return(0);
        }
     }
     // caso em que pelo menos um extremo do segmento esta dentro do quadrante
     if (((raiz->xmin <= xc && xc <= raiz->xmax) && (raiz->ymin <= yc &&
               yc <= raiz->ymax)) || ((raiz->xmin <= xt && xt <= raiz->xmax) &&
                                      (raiz->ymin <= yt && yt <= raiz->ymax))) {
        return(1);
     }
     // neste ponto já podemos supor que o segmento possui pelo menos um extremo
     // externo ao qudrante
     double x0,  // abcissa  do primeiro extremo do lado do quadrante
            y0,  // ordenada do primeiro extremo do lado do quadrante
            x1,  // abcissa  do segundo  extremo do lado do quadrante
            y1,  // ordenada do segundo  extremo do lado do quadrante
            dxl,
            dyl,
            dxs,
            dys,
            det,
            csi,
            eta,
            b1,
            b2;
     // checar interceptação do segmento com a aresta direita do quadrante
     x0 = raiz->xmax;
     x1 = raiz->xmax;
```

```
y0 = raiz->ymin;
y1 = raiz->ymax;
dx1 = x1 - x0;  // aresta vertical -> dx1 = 0
dy1 = y1 - y0;
dxs = xt - xc;
dys = yt - yc;
b1 = x0 - xc;
b2 = y0 - yc;
det = -(dxs*dy1 - dx1*dys);
// lado e segmento paralelos -> dxs = 0
if (det == 0) {
   // testa se aresta e segmento não determinam retas parelalas distintas
   if (xc == x0) {
      // testa se segmento possui um vértice da aresta
      if (((yc <= y0) && (y0 <= yt)) || ((yc <= y1) && (y1 <= yt))) {
         return(1);
      }
   }
}
else {
     csi = (-dy1*b1 + dx1*b2)/det;
     eta = (-dys*b1 + dxs*b2)/det;
     if (0 <= csi && csi <= 1 && 0 <= eta && eta <= 1) {
        return(1);
     }
}
// checar interceptação do segmento com a aresta esquerda do quadrante
x0 = raiz->xmin;
x1 = raiz->xmin;
y0 = raiz->ymin;
y1 = raiz->ymax;
dx1 = x1 - x0;  // aresta vertical -> dx1 = 0
dy1 = y1 - y0;
dxs = xt - xc;
dys = yt - yc;
b1 = x0 - xc;
b2 = y0 - yc;
det = -(dxs*dy1 - dx1*dys);
// lado e segmento paralelos -> dxs = 0
if (det == 0) {
   // testa se aresta e segmento não determinam retas parelalas distintas
   if (xc == x0) {
      // testa se segmento possui um vértice da aresta
      if (((yc <= y0) && (y0 <= yt)) || ((yc <= y1) && (y1 <= yt))) {
         return(1);
      }
   }
}
else {
     csi = (-dy1*b1 + dx1*b2)/det;
     eta = (-dys*b1 + dxs*b2)/det;
     if (0 <= csi && csi <= 1 && 0 <= eta && eta <= 1) {
        return(1);
     }
}
```

```
// checar interceptação do segmento com a aresta inferior do quadrante
x0 = raiz->xmin;
x1 = raiz->xmax;
y0 = raiz->ymin;
y1 = raiz->ymin;
dx1 = x1 - x0;
dy1 = y1 - y0;     // aresta horizontal -> dy1 = 0
dxs = xt - xc;
dys = yt - yc;
b1 = x0 - xc;
b2 = y0 - yc;
det = -(dxs*dy1 - dx1*dys);
// lado e segmento paralelos -> dys = 0
if (det == 0) {
   // testa se aresta e segmento não determinam retas parelalas distintas
   if (yc == y0) {
      // testa se segmento possui um vértice da aresta
      if (((xc <= x0) && (x0 <= xt)) || ((xc <= x1) && (x1 <= xt))) {
         return(1);
      }
   }
}
else {
     csi = (-dy1*b1 + dx1*b2)/det;
     eta = (-dys*b1 + dxs*b2)/det;
     if (0 <= csi && csi <= 1 && 0 <= eta && eta <= 1) {
        return(1);
     }
}
// checar interceptação do segmento com a aresta superior do quadrante
x0 = raiz->xmin;
x1 = raiz->xmax;
y0 = raiz->ymax;
y1 = raiz->ymax;
dx1 = x1 - x0;
dy1 = y1 - y0;     // aresta horizontal -> dy1 = 0
dxs = xt - xc;
dys = yt - yc;
b1 = x0 - xc;
b2 = y0 - yc;
det = -(dxs*dy1 - dx1*dys);
// lado e segmento paralelos -> dys = 0
if (det == 0) {
   // testa se aresta e segmento não determinam retas parelalas distintas
   if (yc == y0) {
      // testa se segmento possui um vértice da aresta
      if (((xc <= x0) && (x0 <= xt)) || ((xc <= x1) && (x1 <= xt))) {
         return(1);
      }
   }
}
else {
     csi = (-dy1*b1 + dx1*b2)/det;
     eta = (-dys*b1 + dxs*b2)/det;
     if (0 <= csi && csi <= 1 && 0 <= eta && eta <= 1) {
```

```
            return(1);
          }
      }
      return(0);
}
/************************************************************************** Fim A */

/*****************************************************************************
 Início B - procedimento para determinação dos quadrantes que formam o box de
            intersecção de um dado vértice
*****************************************************************************/
// esta função armazena na lista "box[4]" de tamanho "tam" os quadrantes que
// compõem o box de intersecção de um dado vértice (x,y)
void detbox(No* raiz, double x, double y) {
     if (raiz != NULL) {
        if (raiz->f0 == NULL) { // testa se "raiz" é folha
           if (raiz->xmin - epsm <= x && x <= raiz->xmax + epsm &&
                              raiz->ymin - epsm <= y && y <= raiz->ymax + epsm) {
              tam++;
              box[tam] = raiz;
           }
        }
        detbox(raiz->f0,x,y);
        detbox(raiz->f1,x,y);
        detbox(raiz->f2,x,y);
        detbox(raiz->f3,x,y);
     }
     return;
}
/************************************************************************** Fim B */

/*****************************************************************************
 Início C - esta função que determina o ponto (xaux,yaux) de interceptação do
            segmento de extremos (xc,yc) -- (xt,yt) com o box de intersecção
            atual, retornando também um ponteiro q para o quadrante em que oorre
            a interceptação
*****************************************************************************/
No* PtoInterceptaBox(double xc,double yc,double xt,double yt,double *xaux,
                                                              double *yaux){
    No* q;
    double x, y,     // coordenadas de interceptação
           x0, y0,
           x1, y1,
           x2, y2,
           x3, y3,  // vértices do quadrante tratado
           dxs, dys,
           dxa, dya,
           det,
           csi,     // parâmetro da parametrização do segmento (xc,yc) - (xt,yt)
           eta,     // parâmetro da parametrização da aresta   (x0,y0) - (x1,y1)
           b1, b2,
           cbx, cby; // coordenadas baricêntricas
    char kk;
    for (kk = 0; kk <= tam; kk++) {
        q = box[kk]; //para cada quadrante do box de intersecção,
```

```
// verifico a intersecção ou não do segmento de extremos (xc,yc)-(xt,yt)
// com cada aresta externa do box de interceptação: eventuais arestas
// internas ao box devem, necessariamente, conter o ponto (xc,yc)

// aresta inferior
x0 = q->xmin;
y0 = q->ymin;
x1 = q->xmax;
y1 = q->ymin;
x2 = q->xmin;
y2 = q->ymax;
x3 = q->xmax;
y3 = q->ymax;
if (!(x0 <= xc && xc <= x1 && yc == y0 && yc == y1)) {
   dxs = xt - xc;
   dys = yt - yc;
   dxa = x1 - x0;
   dya = y1 - y0;
   b1 = x0 - xc;
   b2 = y0 - yc;
   det = -(dxs*dya - dxa*dys);
   // testa se aresta e segmento não são paralelos distintos
   if (det != 0) {
      csi = (-dya*b1 + dxa*b2)/det;
      eta = (-dys*b1 + dxs*b2)/det;
      // 0 < csi -> origem do segmento fora da aresta
      if (0 < csi && csi <= 1 && 0 <= eta && eta <= 1) {
         x = x0 + eta*dxa;
         y = y0 + eta*dya;
         // para cada triângulo, com um lado no segmento (xc,yc)-(x,y)
         // e o outro vértice coincidindo com um dos vértices do
         // quadrante, testo se o baricentro está no interior do semi-
         // plano definido pela aresta e que pertence ao domínio;
         // se um desses baricentros for interior ao domínio, posso
         // retornar o quadrante e passar, por referência, a coordenada
         // da interceptação com o box
         cbx = (x + xc + x0)/3;
         cby = (y + yc + y0)/3;
         if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
            *xaux = x;
            *yaux = y;
            return(q);
         }
         cbx = (x + xc + x1)/3;
         cby = (y + yc + y1)/3;
         if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
            *xaux = x;
            *yaux = y;
            return(q);
         }
         cbx = (x + xc + x2)/3;
         cby = (y + yc + y2)/3;
         if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
            *xaux = x;
            *yaux = y;
```

```
            return(q);
         }
         cbx = (x + xc + x3)/3;
         cby = (y + yc + y3)/3;
         if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
            *xaux = x;
            *yaux = y;
            return(q);
         }
      }
   }
}

// aresta superior
x0 = q->xmin;
y0 = q->ymax;
x1 = q->xmax;
y1 = q->ymax;
x2 = q->xmin;
y2 = q->ymin;
x3 = q->xmax;
y3 = q->ymin;
if (!(x0 <= xc && xc <= x1 && yc == y0 && yc == y1)) {
   dxs = xt - xc;
   dys = yt - yc;
   dxa = x1 - x0;
   dya = y1 - y0;
   b1 = x0 - xc;
   b2 = y0 - yc;
   det = -(dxs*dya - dxa*dys);
   // testa se aresta e segmento não são paralelos distintos
   if (det != 0) {
      csi = (-dya*b1 + dxa*b2)/det;
      eta = (-dys*b1 + dxs*b2)/det;
      // 0 < csi -> origem do segmento fora da aresta
      if (0 < csi && csi <= 1 && 0 <= eta && eta <= 1) {
         x = x0 + eta*dxa;
         y = y0 + eta*dya;
         // para cada triângulo, com um lado no segmento (xc,yc)-(x,y)
         // e o outro vértice coincidindo com um dos vértices do
         // quadrante, testo se o baricentro está no interior do semi-
         // plano definido pela aresta e que pertence ao domínio;
         // se um desses baricentros for interior ao domínio, posso
         // retornar o quadrante e passar, por referência, a coordenada
         // da interceptação com o box
         cbx = (x + xc + x0)/3;
         cby = (y + yc + y0)/3;
         if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
            *xaux = x;
            *yaux = y;
            return(q);
         }
         cbx = (x + xc + x1)/3;
         cby = (y + yc + y1)/3;
         if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
```

```
                *xaux = x;
                *yaux = y;
                return(q);
            }
            cbx = (x + xc + x2)/3;
            cby = (y + yc + y2)/3;
            if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
                *xaux = x;
                *yaux = y;
                return(q);
            }
            cbx = (x + xc + x3)/3;
            cby = (y + yc + y3)/3;
            if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
                *xaux = x;
                *yaux = y;
                return(q);
            }
        }
    }
}

// aresta direita
x0 = q->xmax;
y0 = q->ymin;
x1 = q->xmax;
y1 = q->ymax;
x2 = q->xmin;
y2 = q->ymin;
x3 = q->xmin;
y3 = q->ymax;
if (!(y0 <= yc && yc <= y1 && xc == x0 && xc == x1)) {
    dxs = xt - xc;
    dys = yt - yc;
    dxa = x1 - x0;
    dya = y1 - y0;
    b1 = x0 - xc;
    b2 = y0 - yc;
    det = -(dxs*dya - dxa*dys);
    // testa se aresta e segmento não são paralelos distintos
    if (det != 0) {
        csi = (-dya*b1 + dxa*b2)/det;
        eta = (-dys*b1 + dxs*b2)/det;
        // 0 < csi -> origem do segmento fora da aresta
        if (0 < csi && csi <= 1 && 0 <= eta && eta <= 1) {
            x = x0 + eta*dxa;
            y = y0 + eta*dya;
            // para cada triângulo, com um lado no segmento (xc,yc)-(x,y)
            // e o outro vértice coincidindo com um dos vértices do
            // quadrante, testo se o baricentro está no interior do semi-
            // plano definido pela aresta e que pertence ao domínio;
            // se um desses baricentros for interior ao domínio, posso
            // retornar o quadrante e passar, por referência, a coordenada
            // da interceptação com o box
            cbx = (x + xc + x0)/3;
```

```
            cby = (y + yc + y0)/3;
            if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
               *xaux = x;
               *yaux = y;
               return(q);
            }
            cbx = (x + xc + x1)/3;
            cby = (y + yc + y1)/3;
            if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
               *xaux = x;
               *yaux = y;
               return(q);
            }
            cbx = (x + xc + x2)/3;
            cby = (y + yc + y2)/3;
            if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
               *xaux = x;
               *yaux = y;
               return(q);
            }
            cbx = (x + xc + x3)/3;
            cby = (y + yc + y3)/3;
            if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
               *xaux = x;
               *yaux = y;
               return(q);
            }
         }
      }
   }

   // aresta esquerda
   x0 = q->xmin;
   y0 = q->ymin;
   x1 = q->xmin;
   y1 = q->ymax;
   x2 = q->xmax;
   y2 = q->ymin;
   x3 = q->xmax;
   y3 = q->ymax;
   if (!(y0 <= yc && yc <= y1 && xc == x0 && xc == x1)) {
      dxs = xt - xc;
      dys = yt - yc;
      dxa = x1 - x0;
      dya = y1 - y0;
      b1 = x0 - xc;
      b2 = y0 - yc;
      det = -(dxs*dya - dxa*dys);
      // testa se aresta e segmento não são paralelos distintos
      if (det != 0) {
         csi = (-dya*b1 + dxa*b2)/det;
         eta = (-dys*b1 + dxs*b2)/det;
         // 0 < csi -> origem do segmento fora da aresta
         if (0 < csi && csi <= 1 && 0 <= eta && eta <= 1) {
            x = x0 + eta*dxa;
```

```
                y = y0 + eta*dya;
                // para cada triângulo, com um lado no segmento (xc,yc)-(x,y)
                // e o outro vértice coincidindo com um dos vértices do
                // quadrante, testo se o baricentro está no interior do semi-
                // plano definido pela aresta e que pertence ao domínio;
                // se um desses baricentros for interior ao domínio, posso
                // retornar o quadrante e passar, por referência, a coordenada
                // da interceptação com o box
                cbx = (x + xc + x0)/3;
                cby = (y + yc + y0)/3;
                if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
                   *xaux = x;
                   *yaux = y;
                   return(q);
                }
                cbx = (x + xc + x1)/3;
                cby = (y + yc + y1)/3;
                if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
                   *xaux = x;
                   *yaux = y;
                   return(q);
                }
                cbx = (x + xc + x2)/3;
                cby = (y + yc + y2)/3;
                if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
                   *xaux = x;
                   *yaux = y;
                   return(q);
                }
                cbx = (x + xc + x3)/3;
                cby = (y + yc + y3)/3;
                if (((cby - yc)*(x - xc) + (y - yc)*(xc - cbx)) > 0) {
                   *xaux = x;
                   *yaux = y;
                   return(q);
                }
             }
          }
       }
    }
    return(q);
}
/************************************************************************ Fim C */
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : quadbasica.c
 Descrição : procedimentos relacionados à contrução das quadtrees básicas, por
             vértices e por arestas
*****************************************************************************/

/*****************************************************************************
 Início A - procedimento que constrói uma quadtree básica por vértices, isto é,
            tal que dois vértices quaisquer de contorno não fiquem em um mesmo
            quadrante terminal
OBS.: um vértice qualquer de algum contorno pode estar em vários quadrantes
      terminais, mas, dado um quadrante terminal, este só pode armzenar um único
      vértice, sendo o objetivo desta função garantir exatamente isso
*****************************************************************************/
void CQBV(No *raiz, double x, double y, Vertice *ce, int N, Head *ci, int I) {
     if (x > raiz->xmax || x < raiz->xmin || y > raiz->ymax || y < raiz->ymin) {
        return;
     }
     else {
          flag = 0;
          int ii, jj;
          for (ii = 0; ii <= N; ii++) {
              if (ce[ii].x <= raiz->xmax && ce[ii].x >= raiz->xmin &&
                             ce[ii].y <= raiz->ymax && ce[ii].y >= raiz->ymin) {
                 if (x != ce[ii].x || y != ce[ii].y) {
                    flag = 1;
                 }
              }
          }
          for (ii = 0; ii <= I; ii++) {
              for (jj = 0; jj <= ci[ii].D; jj++) {
                  if (ci[ii].lista[jj].x <= raiz->xmax &&
                                            ci[ii].lista[jj].x >= raiz->xmin &&
                                            ci[ii].lista[jj].y <= raiz->ymax &&
                                             ci[ii].lista[jj].y >= raiz->ymin) {
                     if (x != ci[ii].lista[jj].x || y != ci[ii].lista[jj].y) {
                        flag = 1;
                     }
                  }
              }
          }
          if (flag == 0) {
             return;
          }
          else {
               if (raiz->f0 == NULL) { // testa se raiz é folha
```

```
                AlocaFilhos(raiz);
            }
            CQBV(raiz->f0,x,y,ce,N,ci,I);
            CQBV(raiz->f1,x,y,ce,N,ci,I);
            CQBV(raiz->f2,x,y,ce,N,ci,I);
            CQBV(raiz->f3,x,y,ce,N,ci,I);
        }
    }
    return;
}
/************************************************************************ Fim A*/


/*****************************************************************************
 Início B - função que "refina" uma dada quadtree tal que duas arestas quaisquer
            de contorno não fiquem em um mesmo quadrante terminal (exceção óbvia
            sendo feita ao quadrante onde está a junção de duas arestas
            consecutivas, que, obviamente, é interceptado por tais arestas)
OBS.: uma aresta qualquer de algum contorno pode estar em vários quadrantes
      terminais, mas, dado um quadrante terminal, este só pode interceptar
      uma única aresta (exceto no caso de um vértice comum a arestas adjacentes
      estar dentro do quadrante), sendo o objetivo dessa função garantir
      exatamente isso - esta certeza é indispensável para o processo de
      discretização dos contornos
*****************************************************************************/
void CQB(No *raiz, double xc, double yc, double xt, double yt, Vertice *ce,
                                                    int N, Head *ci, int I) {
    if (!intercepta(raiz,xc,yc,xt,yt)) {
       return;
    }
    else {
        flag = 0;
        // só permito duas arestas interceptarem um mesmo quadrante se
        // isso ocorrer nos extremos do segmento considerado
        if (!(intercepta(raiz,xc,yc,xc,yc) || intercepta(raiz,xt,yt,xt,yt))) {
           int ii, jj;
           for (ii = 0; ii < N; ii++) {
               // verifico se alguma outra aresta intercepta
               // o quadrante considerado
               if (intercepta(raiz,ce[ii].x,ce[ii].y,ce[ii+1].x,ce[ii+1].y)) {
                  // verifico se a aresta que está interceptando o quadrante
                  // não é a aresta inicialmente considerada
                  if (!(xc == ce[ii].x && yc == ce[ii].y &&
                                   xt == ce[ii+1].x && yt == ce[ii+1].y) ) {
                     flag = 1;
                  }
               }
           }
           for (ii = 0; ii <= I; ii++) {
               for (jj = 0; jj < ci[ii].D; jj++) {
                   if (intercepta(raiz,ci[ii].lista[jj].x,ci[ii].lista[jj].y,
                                  ci[ii].lista[jj+1].x,ci[ii].lista[jj+1].y)) {
                      // verifico se a aresta que está interceptando
                      // o quadrante não é a aresta inicialmente considerada
                      if (!(xc == ci[ii].lista[jj].x &&
```

```
                                                    yc == ci[ii].lista[jj].y &&
                                                  xt == ci[ii].lista[jj+1].x &&
                                                yt == ci[ii].lista[jj+1].y) ) {
                                flag = 1;
                             }
                          }
                       }
                    }
                 }
              }
              if (flag == 1) {
                 if (raiz->f0 == NULL) { // testa se raiz é folha
                      AlocaFilhos(raiz);
                 }
              }
              if (raiz->f0 != NULL) { // testa se raiz não é folha
                 CQB(raiz->f0,xc,yc,xt,yt,ce,N,ci,I);
                 CQB(raiz->f1,xc,yc,xt,yt,ce,N,ci,I);
                 CQB(raiz->f2,xc,yc,xt,yt,ce,N,ci,I);
                 CQB(raiz->f3,xc,yc,xt,yt,ce,N,ci,I);
              }
        }
        return;
}
/************************************************************************** Fim B*/
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : restante.c
 Descrição : procedimento que garante que a quadtree atinja os níveis mínimos
             exigidos pelo usuário e especificados no arquivo ARQFCC definido
*****************************************************************************/

/*****************************************************************************
 Início A - função que constrói a quadtree restante, satisfazendo os níveis de
            refinamento exigidos nos vértices e arestas de contorno ou controle
*****************************************************************************/
void CQR(No *raiz, double xc, double yc, double xt, double yt, int peso) {
     if (!intercepta(raiz,xc,yc,xt,yt)) {
        return;
     }
     else {
          if (peso <= raiz->niv) {
             return;
          }
          else {
               if (raiz->f0 == NULL) { // testa se raiz é folha
                  AlocaFilhos(raiz);
               }
               CQR(raiz->f0,xc,yc,xt,yt,peso);
               CQR(raiz->f1,xc,yc,xt,yt,peso);
               CQR(raiz->f2,xc,yc,xt,yt,peso);
               CQR(raiz->f3,xc,yc,xt,yt,peso);
          }
     }
     return;
}
/***************************************************************************** Fim A*/
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema   : QUADMESH 2D - gerador de malhas quadtree
 Módulo    : savedge.c
 Descrição : coleção de procedimentos que percorrem a quadtree, salvando
             informações convenientes da mesma, em cada etapa da execução, para
             representação gráfica das etapas de construção da malha quadtree
*****************************************************************************/

/*****************************************************************************
 Início A - conjunto de funções que percorre uma quadtree, salvando em arquivo
            os extremos dos quadrantes terminais
*****************************************************************************/
// função recursiva para percorrer a quadtree e salvar em arquivo os extremos
// dos quadrantes terminais, complementando a função "PercursoImprimeQ"
void AUXPercursoImprimeQ(No *raiz, FILE *entrada) {
     if (raiz != NULL) {
        if (raiz->f0 == NULL) { // testa se raiz é folha
           fprintf(entrada, "%le %le %le %le\n", raiz->xmin, raiz->xmax,
                                                 raiz->ymin, raiz->ymax);
        }
        AUXPercursoImprimeQ(raiz->f0,entrada);
        AUXPercursoImprimeQ(raiz->f1,entrada);
        AUXPercursoImprimeQ(raiz->f2,entrada);
        AUXPercursoImprimeQ(raiz->f3,entrada);
     }
     return;
}

// função que percorre a quadtree e salva em arquivo os extremos dos quadrantes
void PercursoImprimeQ(cadeia ARQFQ, No* raiz) {
     FILE *entrada;
     entrada = AbreArquivo(ARQFQ,MODOw);
     AUXPercursoImprimeQ(raiz,entrada);
     FechaArquivo(entrada);
     return;
}
/***************************************************************************** Fim A*/

/*****************************************************************************
 Início B - procedimento para guardar as arestas dos quadrantes de contorno,
            já devidamente recortadas
*****************************************************************************/
// função recursiva para percorrer a quadtree e salvar em arquivo os quadrantes
// que discretizam o contorno, complementando a função "PercursoImprimeQCD"
void AUXPercursoImprimeCDQ(No *raiz, FILE *entrada) {
     if (raiz != NULL) {
        if (raiz->f0 == NULL) { // testa se raiz é folha
```

```
            if (raiz->pos >= 0) { // testa se o quadrante é de contorno
               char ii;
               for (ii = 0; ii < raiz->pos; ii++) {
                   fprintf(entrada, "%le %le %le %le\n", raiz->xelem[ii],
                          raiz->xelem[ii+1], raiz->yelem[ii], raiz->yelem[ii+1]);
               }
            }
         }
         AUXPercursoImprimeCDQ(raiz->f0,entrada);
         AUXPercursoImprimeCDQ(raiz->f1,entrada);
         AUXPercursoImprimeCDQ(raiz->f2,entrada);
         AUXPercursoImprimeCDQ(raiz->f3,entrada);
      }
      return;
}


// função que percorre a quadtree e salva em arquivo os quadrantes de contorno
void PercursoImprimeCDQ(cadeia ARQFQ, No* raiz) {
      FILE *entrada;
      entrada = AbreArquivo(ARQFQ,MODOw);
      AUXPercursoImprimeCDQ(raiz,entrada);
      FechaArquivo(entrada);
      return;
}
/************************************************************************** Fim B*/

/*****************************************************************************
 Início C - procedimento que salva todas as arestas dos quadrantes terminais
            da malha quadtree
*****************************************************************************/
// função recursiva para percorrer a quadtree e salvar em arquivo os elementos
// que discretizam o domínio, complementando a função "PercursoImprimeMQ"
void AUXPercursoImprimeMQ(No *raiz, FILE *entrada) {
      if (raiz != NULL) {
         if (raiz->f0 == NULL) { // testa se raiz é folha
            if (raiz->pos != -3) { // testa se o elemento não é externo
               char ii;
               if (raiz->pos >=0) {
                  for (ii = 0; ii < raiz->pos; ii++) {
                      fprintf(entrada, "%le %le %le %le\n", raiz->xelem[ii],
                          raiz->xelem[ii+1], raiz->yelem[ii], raiz->yelem[ii+1]);
                  }
               } else {
                      fprintf(entrada, "%le %le %le %le\n", raiz->xmin,
                                       raiz->xmax, raiz->ymin, raiz->ymin);
                      fprintf(entrada, "%le %le %le %le\n", raiz->xmin,
                                       raiz->xmax, raiz->ymax, raiz->ymax);
                      fprintf(entrada, "%le %le %le %le\n", raiz->xmin,
                                       raiz->xmin, raiz->ymin, raiz->ymax);
                      fprintf(entrada, "%le %le %le %le\n", raiz->xmax,
                                       raiz->xmax, raiz->ymin, raiz->ymax);

               }
```

```
            }
        }
        AUXPercursoImprimeMQ(raiz->f0,entrada);
        AUXPercursoImprimeMQ(raiz->f1,entrada);
        AUXPercursoImprimeMQ(raiz->f2,entrada);
        AUXPercursoImprimeMQ(raiz->f3,entrada);
    }
    return;
}

// função que percorre a quadtree e salva em arquivo os quadrantes não externos
void PercursoImprimeMQ(cadeia ARQFQ, No* raiz) {
    FILE *entrada;
    entrada = AbreArquivo(ARQFQ,MODOw);
    AUXPercursoImprimeMQ(raiz,entrada);
    FechaArquivo(entrada);
    return;
}
/************************************************************************** Fim C*/
```

```
/*****************************************************************************
 Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
 Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
 Data   : 23/11/2007
*****************************************************************************/

/*****************************************************************************
 Sistema           : QUADMESH 2D - gerador de malhas quadtree
 Módulo Principal  : quadmesh2D.c
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "base.h"      // <-- definição dos parâmetros de execução
#include "arqmanip.h"
#include "interceptos.h"
#include "quadbasica.h"
#include "restante.h"
#include "savedge.h"
#include "balanc.h"
#include "discont.h"
#include "extint.h"

int main() {
    int ii, jj, // indexadores para andar nas listas e sub-istas
        kk;     // indexador auxiliar
    int N,  // número de elementos na lista de vértices, indexados de 0,1,...,N
        I,  // número de listas de ciclos internos, indexadas de 0,1,...,N
        P;  // número de listas de poligonais de controle, indexadas de 0,1,...N
    Vertice *ce; // vetor de vértices do contorno externo
    Head *ci,    // vetor de vetores de vértices de contorno interno
         *pc;    // vetor de vetores de vértice de poligonais de controle
    ListasIn(&ce,&N,&ci,&I,&pc,&P);
    No *quad;    // raiz da quadtree
    clock_t ti;
    double xmin,
           xmax,
           ymin,
           ymax,   // extremos do contorno externo
           xmedio, // abcissa  média do contorno externo
           ymedio, // ordenada média do contorno externo
           lado;   // lado do quadrante universo

    xmin = ce[0].x;  // valores de partida para encontrar os extremos
    xmax = ce[0].x;  // do quadrante universo
    ymin = ce[0].y;
    ymax = ce[0].y;
    // inicio - impressão dos dados de entrada para verificação
    printf("\n\t\tQUADMESH 2D - Gerador de Malhas Quadtree (versao 1.0)\n");
    printf("\nAutor : Fernando Pacanelli Martins");
    printf("\n\nProcessamento iniciado...\n");
```

```
ti = clock();
printf("\n\t\tDados relativos ao dominio");
printf("\nContorno externo\n");
printf("   x          y         pv  pa\n\n");
for (ii = 0; ii <= N; ii++) {
    printf("%10.5lf %10.5lf %3i %3i\n",ce[ii].x, ce[ii].y, ce[ii].pv,
                                                        ce[ii].pa);
    // início - determina extremos do contorno externo
    xmin = (xmin < ce[ii].x) ? xmin : ce[ii].x;
    xmax = (xmax > ce[ii].x) ? xmax : ce[ii].x;
    ymin = (ymin < ce[ii].y) ? ymin : ce[ii].y;
    ymax = (ymax > ce[ii].y) ? ymax : ce[ii].y;
    // fim    - determina extremos do contorno externo
}
for (ii = 0; ii <= I; ii++) {
    printf("\nCiclo interno: %i\n", I);
    printf("   x          y         pv  pa\n\n");
    for (jj = 0; jj <= ci[ii].D; jj++) {
        printf("%10.5lf %10.5lf %3i %3i\n", ci[ii].lista[jj].x,
            ci[ii].lista[jj].y, ci[ii].lista[jj].pv, ci[ii].lista[jj].pa);
    }
}
for (ii = 0; ii <= P; ii++) {
    printf("\nPoligonal de controle: %i\n", pc[ii].D);
    printf("   x          y         pv  pa\n\n");
    for (jj = 0; jj <= pc[ii].D; jj++) {
        printf("%10.5lf %10.5lf %3i %3i\n", pc[ii].lista[jj].x,
            pc[ii].lista[jj].y, pc[ii].lista[jj].pv, pc[ii].lista[jj].pa);
    }
}
// fim   - impressão dos dados de entrada para verificação

// início - definição dos extremos do quadrante universo
quad = AlocaNoQ();
quad->niv = 1;
lado = ((xmax - xmin) > (ymax - ymin)) ? (xmax - xmin) : (ymax - ymin);
xmedio = xmin + 0.5*(xmax - xmin);
ymedio = ymin + 0.5*(ymax - ymin);
quad->xmin = xmedio - 0.5*lado;
quad->xmax = xmedio + 0.5*lado;
quad->ymin = ymedio - 0.5*lado;
quad->ymax = ymedio + 0.5*lado;
// fim    - definição dos extremos do quadrante universo

// início - construção da quadtree básica por vértices
for (ii = 0; ii <= N; ii++) {
    CQBV(quad,ce[ii].x,ce[ii].y,ce,N,ci,I);
}


for (ii = 0; ii <= I; ii++) {
    for (jj = 0; jj <= ci[ii].D; jj++) {
        CQBV(quad,ci[ii].lista[jj].x,ci[ii].lista[jj].y,ce,N,ci,I);
    }
}
```

```
PercursoImprimeQ(ARQFQA,quad); // os quadantes terminais da quadtree básica
                               // por vértices são salvos em arquivos
// fim   - construção da quadtree básica por vértices

// início - construção da quadtree restante
for (ii = 0; ii < N; ii++) {
    // vértice "ii"
    CQR(quad,ce[ii].x,ce[ii].y,ce[ii].x,ce[ii].y,ce[ii].pv);
    // aresta "(ii):(ii+1)"
    CQR(quad,ce[ii].x,ce[ii].y,ce[ii+1].x,ce[ii+1].y,ce[ii].pa);
}
for (ii = 0; ii <= I; ii++) {
    for (jj = 0; jj < ci[ii].D; jj++) {
        // vértice "ii"
        CQR(quad,ci[ii].lista[jj].x,ci[ii].lista[jj].y,ci[ii].lista[jj].x,
                                    ci[ii].lista[jj].y,  ci[ii].lista[jj].pv);
        // aresta "(ii):(ii+1)"
        CQR(quad,ci[ii].lista[jj].x,ci[ii].lista[jj].y,ci[ii].lista[jj+1].x,
                                    ci[ii].lista[jj+1].y,ci[ii].lista[jj].pa);
    }
}
for (ii = 0; ii <= P; ii++) {
    for (jj = 0; jj < pc[ii].D; jj++) {
        // vértice "ii"
        CQR(quad,pc[ii].lista[jj].x,pc[ii].lista[jj].y,pc[ii].lista[jj].x,
                                    pc[ii].lista[jj].y,  pc[ii].lista[jj].pv);
        // aresta "(ii):(ii+1)"
        CQR(quad,pc[ii].lista[jj].x,pc[ii].lista[jj].y,pc[ii].lista[jj+1].x,
                                    pc[ii].lista[jj+1].y,pc[ii].lista[jj].pa);
    }
    jj = pc[ii].D;
    CQR(quad,pc[ii].lista[jj].x,pc[ii].lista[jj].y,pc[ii].lista[jj].x,
                                pc[ii].lista[jj].y,  pc[ii].lista[jj].pv);
}
PercursoImprimeQ(ARQFQB,quad); // os quadantes terminais da quadtree
                               // restante são salvos em arquivo
// fim    - construção da quadtree restante

// início - construção da quadtree básica por arestas
for (ii = 0; ii < N; ii++) {
    CQB(quad,ce[ii].x,ce[ii].y,ce[ii+1].x,ce[ii+1].y,ce,N,ci,I);
}

for (ii = 0; ii <= I; ii++) {
    for (jj = 0; jj < ci[ii].D; jj++) {
        CQB(quad,ci[ii].lista[jj].x,ci[ii].lista[jj].y,ci[ii].lista[jj+1].x,
                                    ci[ii].lista[jj+1].y,ce,N,ci,I);
    }
}

PercursoImprimeQ(ARQFQC,quad); // os quadantes terminais da quadtree básica
                               // por arestas são salvos em arquivos
// fim   - construção da quadtree básica por arestas
```

```
// início - balanceamento da quadtree segundo a tolerância dada
flag = 1;
while (flag == 1) {
      flag = 0;
      ChecaBalanceamento(quad,quad);
}
PercursoImprimeQ(ARQFQD,quad); // os quadantes terminais da quadtree
                               // balanceada são salvos em arquivo
// fim    - balanceamento da quadtree segundo a tolerância dada

double xc, yc,
       xt, yt,      // coordenadas extremas de uma aresta de contorno
       xaux, yaux,  // coordenada de início de uma aresta atualizada
       xa, ya,
       cbx, cby;
No *q, *noaux;;
// início - processo de discretização das fronteiras
// início - processo de discretização do contorno externo
for (ii = 0; ii < N; ii++) {
    xc = ce[ii].x;
    yc = ce[ii].y;
    xt = ce[ii+1].x;
    yt = ce[ii+1].y;
    do {
       tam = -1;
       detbox(quad,xc,yc); // determinação do box de intersecção
       flag = 1;
       // pela construção da "quadtree básica", o teste if dentro deste
       // primeiro laço for só será satisfeito no último box de intersecção
       // para a aresta considerada
       for (kk = 0; kk <= tam; kk++) {
           q = box[kk];
           if (q->xmin <= xt && xt <= q->xmax && q->ymin <= yt &&
                                                        yt <= q->ymax) {
              // teste para verificar se o box é de contorno ou exterior

              // (x0,y0) - vértice sudoeste do quadrante
              xa = q->xmin;
              ya = q->ymin;
              cbx = (xt + xc + xa)/3;
              cby = (yt + yc + ya)/3;
              if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
                 flag = 0; // q é quadrante de contorno E contém (xt,yt)
              }

              // (x0,y0) - vértice sudeste do quadrante
              xa = q->xmax;
              ya = q->ymin;
              cbx = (xt + xc + xa)/3;
              cby = (yt + yc + ya)/3;
              if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
                 flag = 0; // q é quadrante de contorno E contém (xt,yt)
              }
```

```
            // (x0,y0) - vértice noroeste do quadrante
            xa = q->xmin;
            ya = q->ymax;
            cbx = (xt + xc + xa)/3;
            cby = (yt + yc + ya)/3;
            if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
               flag = 0; // q é quadrante de contorno E contém (xt,yt)
            }

            // (x0,y0) - vértice nordeste do quadrante
            xa = q->xmax;
            ya = q->ymax;
            cbx = (xt + xc + xa)/3;
            cby = (yt + yc + ya)/3;
            if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
               flag = 0; // q é quadrante de contorno E contém (xt,yt)
            }

            if (flag == 0) {
               if (q->xelem == NULL || q->yelem == NULL) {
                  q->xelem = (double *)malloc(8*sizeof(double));
                  q->yelem = (double *)malloc(8*sizeof(double));
               }
               (q->pos)++;
               q->xelem[q->pos] = xc;
               q->yelem[q->pos] = yc;
               // se o ponto (xt,yt) estiver sobre a fronteira do
               // quadrante apontado por "q" e a próxima interceptaçãodo
               // segmento não ocorrer no quadrante "q", preciso armazenar
               // o ponto (xt,yt)
               if (xt == q->xmin || xt == q->xmax || yt == q->ymin ||
                                                     yt == q->ymax) {
                  tam = -1;
                  detbox(quad,xt,yt); // previsão do próximo box
                  if (ii + 1 == N) {
                     xaux = ce[1].x;
                     yaux = ce[1].y;
                  }
                  else {
                       xaux = ce[ii+2].x;
                       yaux = ce[ii+2].y;
                  }
                  noaux = PtoInterceptaBox(xt,yt,xaux,yaux,&xaux,&yaux);
                  if (q != noaux) {
                     (q->pos)++;
                     q->xelem[q->pos] = xt;
                     q->yelem[q->pos] = yt;
                  }
               }
               kk = tam;
            }
         }
      }

      if (flag == 1) { // o ponto (xt,yt) é externo ao box
```

```
            q = PtoInterceptaBox(xc,yc,xt,yt,&xaux,&yaux);
            if (q->xelem == NULL || q->yelem == NULL) {
               q->xelem = (double *)malloc(8*sizeof(double));
               q->yelem = (double *)malloc(8*sizeof(double));
            }
            (q->pos)++;
            q->xelem[q->pos] = xc;
            q->yelem[q->pos] = yc;
            (q->pos)++;
            q->xelem[q->pos] = xaux;
            q->yelem[q->pos] = yaux;
            xc = xaux;
            yc = yaux;
         }
      } while (flag == 1);
   }
   // quadrante de contorno para o primeiro vértice da lista, caso ele
   // possua três elementos, precisamo colocar o primeiro ponto que está na
   // lista (xelem,yelem) na última posição válida da mesma, afim de que os
   // elementos dessa lista estejam na mesma sequência que a definida pela
   // orientação do contorno
   ii = 0;
   xc = ce[ii].x;
   yc = ce[ii].y;
   xt = ce[ii+1].x;
   yt = ce[ii+1].y;
   tam = -1;
   detbox(quad,xc,yc); // determinação do box de intersecção
   q = PtoInterceptaBox(xc,yc,xt,yt,&xaux,&yaux);
   if (q->pos == 2) {
      q->xelem[3] = q->xelem[2];
      q->xelem[2] = q->xelem[1];
      q->xelem[1] = q->xelem[0];
      q->xelem[0] = q->xelem[3];

      q->yelem[3] = q->yelem[2];
      q->yelem[2] = q->yelem[1];
      q->yelem[1] = q->yelem[0];
      q->yelem[0] = q->yelem[3];
   }
   // fim    - processo de discretização do contorno externo

   // início - processo de discretização dos contornos internos
   for (ii = 0; ii <= I; ii++) {
       for (jj = 0; jj < ci[ii].D; jj++) {
           xc = ci[ii].lista[jj].x;
           yc = ci[ii].lista[jj].y;
           xt = ci[ii].lista[jj+1].x;
           yt = ci[ii].lista[jj+1].y;
           do {
               tam = -1;
               detbox(quad,xc,yc); // determinação do box de intersecção
               flag = 1;
               // pela construção da "quadtree básica", o teste if dentro deste
               // primeiro laço for só será satisfeito no último box de
```

```
// intersecção para a aresta considerada
for (kk = 0; kk <= tam; kk++) {
    q = box[kk];
    if (q->xmin <= xt && xt <= q->xmax && q->ymin <= yt &&
                                                yt <= q->ymax) {
       // teste para verificar se o box é de contorno
       // ou exterior

       // (x0,y0) - vértice sudoeste do quadrante
       xa = q->xmin;
       ya = q->ymin;
       cbx = (xt + xc + xa)/3;
       cby = (yt + yc + ya)/3;
       if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
          flag = 0;// q é quadrante de contorno E contém (xt,yt)
       }

       // (x0,y0) - vértice sudeste do quadrante
       xa = q->xmax;
       ya = q->ymin;
       cbx = (xt + xc + xa)/3;
       cby = (yt + yc + ya)/3;
       if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
          flag = 0;// q é quadrante de contorno E contém (xt,yt)
       }

       // (x0,y0) - vértice noroeste do quadrante
       xa = q->xmin;
       ya = q->ymax;
       cbx = (xt + xc + xa)/3;
       cby = (yt + yc + ya)/3;
       if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
          flag = 0;// q é quadrante de contorno E contém (xt,yt)
       }

       // (x0,y0) - vértice nordeste do quadrante
       xa = q->xmax;
       ya = q->ymax;
       cbx = (xt + xc + xa)/3;
       cby = (yt + yc + ya)/3;
       if (((cby - yc)*(xt - xc) + (yt - yc)*(xc - cbx)) > 0) {
          flag = 0; // q é quadrante de contorno E contém (xt,yt)
       }

       if (flag == 0) {
          if (q->xelem == NULL || q->yelem == NULL) {
             q->xelem = (double *)malloc(8*sizeof(double));
             q->yelem = (double *)malloc(8*sizeof(double));
          }
          (q->pos)++;
          q->xelem[q->pos] = xc;
          q->yelem[q->pos] = yc;
          // se o ponto (xt,yt) estiver sobre a fronteira do
          // quadrante apontado por "q" e a próxima
          // interceptaçãodo segmento não ocorrer no quadrante
```

```
                        // "q", preciso armazenar o ponto (xt,yt)
                        if (xt == q->xmin || xt == q->xmax || yt == q->ymin ||
                                                              yt == q->ymax) {
                           tam = -1;
                           detbox(quad,xt,yt); // previsão do próximo box
                           if (jj + 1 == ci[ii].D) {
                              xaux = ci[ii].lista[1].x;
                              yaux = ci[ii].lista[1].y;
                           }
                           else {
                                xaux = ci[ii].lista[jj+2].x;
                                yaux = ci[ii].lista[jj+2].y;
                           }
                           noaux =
PtoInterceptaBox(xt,yt,xaux,yaux,&xaux,&yaux);
                           if (q != noaux) {
                              (q->pos)++;
                              q->xelem[q->pos] = xt;
                              q->yelem[q->pos] = yt;
                           }

                        }
                        kk = tam;
                     }
                  }
               }
               if (flag == 1) { // o ponto (xt,yt) é externo ao box
                  q = PtoInterceptaBox(xc,yc,xt,yt,&xaux,&yaux);
                  if (q->xelem == NULL || q->yelem == NULL) {
                     q->xelem = (double *)malloc(8*sizeof(double));
                     q->yelem = (double *)malloc(8*sizeof(double));
                  }
                  (q->pos)++;
                  q->xelem[q->pos] = xc;
                  q->yelem[q->pos] = yc;
                  (q->pos)++;
                  q->xelem[q->pos] = xaux;
                  q->yelem[q->pos] = yaux;
                  xc = xaux;
                  yc = yaux;
               }
            } while (flag == 1);
         }
         // quadrante de contorno para o primeiro vértice da lista, caso ele
         // possua três elementos, precisamo colocar o primeiro ponto que esta na
         // lista (xelem,yelem) na última posição válida da mesma, afim de que os
         // elementos dessa lista estejam na mesma sequência que a definida pela
         // orientação do contorno
         jj=0;
         xc = ci[ii].lista[jj].x;
         yc = ci[ii].lista[jj].y;
         xt = ci[ii].lista[jj+1].x;
         yt = ci[ii].lista[jj+1].y;
         tam = -1;
         detbox(quad,xc,yc); // determinação do box de intersecção
```

```
        q = PtoInterceptaBox(xc,yc,xt,yt,&xaux,&yaux);
        if (q->pos == 2) {
           q->xelem[3] = q->xelem[2];
           q->xelem[2] = q->xelem[1];
           q->xelem[1] = q->xelem[0];
           q->xelem[0] = q->xelem[3];

           q->yelem[3] = q->yelem[2];
           q->yelem[2] = q->yelem[1];
           q->yelem[1] = q->yelem[0];
           q->yelem[0] = q->yelem[3];
        }
    }
    // fim    - processo de discretização dos contornos internos

    DefElemContorno(quad);
    PercursoImprimeCDQ(ARQFQE,quad); //imprime os contornos discretizados
    // fim    - processo de discretização das fronteiras

    // início - determinação da posição relativa dos demais quadrantes
    double xr, yr; // coordenada do ponto remoto
    xr = 1 + quad->xmax;
    yr = quad->ymin;    // o ponto remoto deve ser externo ao quadrante universo
    CassaPosRel(quad,xr,yr,ce,N,ci,I);
    PercursoImprimeMQ(ARQFQF,quad); //imprime os contornos discretizados
    // fim    - determinação da posição relativa dos demais quadrantes


    printf("\n\nProcessamento concluido com sucesso em %lg segundos.\n\n\n\t\t",
                                          (double)(clock()-ti)/CLOCKS_PER_SEC);
    system("pause");
    return;
}
```

```
%{
Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                 Mestrando em Matemática Aplicada e Computacional
                 Mecânica dos Fluidos Computacional
Data       : 23/11/2007
Sistema    : QUADMESH 2D - gerador de malhas quadtree
Módulo     : quatree.m
Instruções: durante a execução do QUADMESH2D.EXE são criados arquivos que
            possibilitam ilustrar cada uma das etapas da criação de uma
            malha quadtree; assim, a função deste script é ler tais
            arquivos, bem como o arquivo que define o domínio de entrada
            para o gerador de malha, fazendo as devidas representações
            gráficas
IMPORTANTE: o nome dos arquivos físicos empregados devem ser especificados
            igualmente tanto no QUADMESH2D como neste script para que
            a representação gráfica seja coerente com os dados de entrada
            e saída do gerador
%}
clc
clear

% Parte 0 - parâmetros de entrada
ex = num2str(0);  % define o tipo do exemplo
res  = '-r150';   % resolução das figuras de saída
disp = '-dbmp';   % formato das figuras de saída
ARQFCC = strcat('exemplo',ex,'.txt');
ARQFQA = strcat('quadexA',ex,'.txt');
ARQFQB = strcat('quadexB',ex,'.txt');
ARQFQC = strcat('quadexC',ex,'.txt');
ARQFQD = strcat('quadexD',ex,'.txt');
ARQFQE = strcat('quadexE',ex,'.txt');
ARQFQF = strcat('quadexF',ex,'.txt');
% FIM - Parte 0

% Parte 1 - Representação dos contornos da região e poligonais de controle
figure(1);
hold on;
RepresentaContornoPeso;
xlabel('x');
ylabel('y');
title(['Domínio e poligonais de controle']);
grid on;
print(disp,res,strcat(ex,'ACCcntcntr'));
% FIM - Parte 1

% Parte 2 - Representação da quadtree básica por vértices
figure(2);
hold on;
QA = load(ARQFQA, '-ascii');
[m,n] = size(QA);
for ii = 1:1:m
    plot([QA(ii,1),QA(ii,2)],[QA(ii,3),QA(ii,3)],'linewidth', 1);
    plot([QA(ii,1),QA(ii,2)],[QA(ii,4),QA(ii,4)],'linewidth', 1);
```

```
    plot([QA(ii,1),QA(ii,1)],[QA(ii,3),QA(ii,4)],'linewidth', 1);
    plot([QA(ii,2),QA(ii,2)],[QA(ii,3),QA(ii,4)],'linewidth', 1);
end
RepresentaContorno;
xlabel('x');
ylabel('y');
title(['Quadtree básica por vértices']);
print(disp,res,strcat(ex,'Aqdtbasicav'));
% FIM - Parte 2

% Parte 3 - Representação da quadtree restante
figure(3);
hold on;
QB = load(ARQFQB, '-ascii');
[m,n] = size(QB);
for ii = 1:1:m
    plot([QB(ii,1),QB(ii,2)],[QB(ii,3),QB(ii,3)],'linewidth', 1);
    plot([QB(ii,1),QB(ii,2)],[QB(ii,4),QB(ii,4)],'linewidth', 1);
    plot([QB(ii,1),QB(ii,1)],[QB(ii,3),QB(ii,4)],'linewidth', 1);
    plot([QB(ii,2),QB(ii,2)],[QB(ii,3),QB(ii,4)],'linewidth', 1);
end
RepresentaContorno;
xlabel('x');
ylabel('y');
title(['Quadtree restante']);
print(disp,res,strcat(ex,'Bqdtrestante'));
% FIM - Parte 3

% Parte 4 - Representação da quadtree básica por arestas
figure(4);
hold on;
QC = load(ARQFQC, '-ascii');
[m,n] = size(QC);
for ii = 1:1:m
    plot([QC(ii,1),QC(ii,2)],[QC(ii,3),QC(ii,3)],'linewidth', 1);
    plot([QC(ii,1),QC(ii,2)],[QC(ii,4),QC(ii,4)],'linewidth', 1);
    plot([QC(ii,1),QC(ii,1)],[QC(ii,3),QC(ii,4)],'linewidth', 1);
    plot([QC(ii,2),QC(ii,2)],[QC(ii,3),QC(ii,4)],'linewidth', 1);
end
RepresentaContorno;
xlabel('x');
ylabel('y');
title(['Quadtree básica por arestas']);
print(disp,res,strcat(ex,'Cqdtbasicaa'));
% FIM - Parte 4

% Parte 5 - Representação da quadtree balanceada
figure(5);
hold on;
QD = load(ARQFQD, '-ascii');
[m,n] = size(QD);
for ii = 1:1:m
    plot([QD(ii,1),QD(ii,2)],[QD(ii,3),QD(ii,3)],'linewidth', 1);
    plot([QD(ii,1),QD(ii,2)],[QD(ii,4),QD(ii,4)],'linewidth', 1);
```

```
    plot([QD(ii,1),QD(ii,1)],[QD(ii,3),QD(ii,4)],'linewidth', 1);
    plot([QD(ii,2),QD(ii,2)],[QD(ii,3),QD(ii,4)],'linewidth', 1);
end
RepresentaContorno;
xlabel('x');
ylabel('y');
title(['Quadtree balanceada']);
print(disp,res,strcat(ex,'Dqdtbalanceada'));
% FIM - Parte 5

% Parte 6 - Representação dos elementos de contorno
figure(6);
hold on;
QE = load(ARQFQE, '-ascii');
[m,n] = size(QE);

for ii = 1:1:m
    plot([QE(ii,1),QE(ii,2)],[QE(ii,3),QE(ii,4)],'b','linewidth', 1);
end
RepresentaContorno;
grid on;
xlabel('x');
ylabel('y');
title(['Elementos de contorno']);
print(disp,res,strcat(ex,'Eqdtelemcont'));
% FIM - Parte 6

% Parte 7 - Representação da malha quadtree
figure(7);
hold on;
QF = load(ARQFQF, '-ascii');
[m,n] = size(QF);

for ii = 1:1:m
    plot([QF(ii,1),QF(ii,2)],[QF(ii,3),QF(ii,4)],'b','linewidth', 1);
end
RepresentaContorno;
grid on;
xlabel('x');
ylabel('y');
title(['Malha quadtree']);
print(disp,res,strcat(ex,'Fmalhaqtd'));
% FIM - Parte 7
```

```
%{
Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
Data       : 23/11/2007
Sistema    : QUADMESH 2D - gerador de malhas quadtree
Módulo     : RepresentaContorno.m
Descrição : Instruções para representar o contorno e poligonais de controle
            do domínio, sem mostrar o nível de refinamento exigido por cada
            ente geométrico.
%}
fid = fopen(ARQFCC,'r');  % -> arquivo de definição do domínio

% Parte A - plotagem do contorno externo
N  = fscanf(fid,'%d', 1); % -> número de vértices do ciclo externo
ce = zeros(N+1,4);        % -> aloca-se um vértice a mais para que se possa
                          % repetir o primeiro vértice na última posição,
                          % de forma a "fechar" o ciclo que define o
                          % contorno externo

for ii = 1:1:N
    ce(ii,1) = fscanf(fid, '%f ',1); % -> abcissa  do vértice
    ce(ii,2) = fscanf(fid, '%f ',1); % -> ordenada do vértice
    ce(ii,3) = fscanf(fid, '%d ',1); % -> nível exigido pelo vértice
    ce(ii,4) = fscanf(fid, '%d ',1); % -> nível exigido pela aresta que se
                                     % inicia neste vértice
end
N = N + 1;          % -> replicação do primeiro vértice na última posição
ce(N,1) = ce(1,1);  % para facilitar a plotagem e fechar o ciclo
ce(N,2) = ce(1,2);
ce(N,3) = ce(1,3);
ce(N,4) = ce(1,4);
plot(ce(:,1),ce(:,2),'.-r','linewidth', 1.3); %plotagem do contorno externo
xmin = min(ce(:,1));
ymin = min(ce(:,2));
xmax = max(ce(:,1));
ymax = max(ce(:,2));
lado = max(xmax-xmin,ymax-ymin) + 1; % -> lado do quadrante universo
xmedio = 0.5*(xmin + xmax);
ymedio = 0.5*(ymin + ymax);
axis([xmedio-0.5*lado,xmedio+0.5*lado,ymedio-0.5*lado,ymedio+0.5*lado]);
clear ce N xmin xmax ymin ymax lado xmedio ymedio % -> elimina as variáveis
                                                  % usadas nesse bloco
% FIM Parte A

% Parte B - plotagem dos contornos internos
I  = fscanf(fid,'%d', 1);   % -> número de ciclos internos
for ii = 1:1:I
    D = fscanf(fid,'%d', 1);  % -> número de vértices em cada ciclo interno
    ci = zeros(D+1,4);
    for jj = 1:1:D
        ci(jj,1) = fscanf(fid, '%f ',1);  % -> abcissa  do vértice
        ci(jj,2) = fscanf(fid, '%f ',1);  % -> ordenada do vértice
```

```
        ci(jj,3) = fscanf(fid, '%d ',1);  % -> nível exigido pelo vértice
        ci(jj,4) = fscanf(fid, '%d ',1);  % -> nível exigido pela aresta
                                          % que se inicia neste vértice
    end
    D = D + 1;          % -> replicação do primeiro vértice na última posição
    ci(D,1) = ci(1,1);% para facilitar a plotagem e fechar o ciclo
    ci(D,2) = ci(1,2);
    ci(D,3) = ci(1,3);
    ci(D,4) = ci(1,4);
    plot(ci(:,1),ci(:,2),'.-r','linewidth', 1.3);
end
clear ci I D   % -> elimina as variáveis usadas nesse bloco
% FIM Parte B


% Parte C - plotagem das poligonais de controle
P  = fscanf(fid,'%d', 1);     % -> número de poligonais de controle
for ii = 1:1:P
    C = fscanf(fid,'%d', 1); % -> número de vértices em cada poligonal
    pc = zeros(C,4);
    for jj = 1:1:C
        pc(jj,1) = fscanf(fid, '%f ',1);  % -> abcissa  do vértice
        pc(jj,2) = fscanf(fid, '%f ',1);  % -> ordenada do vértice
        pc(jj,3) = fscanf(fid, '%d ',1);  % -> nível exigido pelo vértice
        pc(jj,4) = fscanf(fid, '%d ',1);  % -> nível exigido pela aresta
                                          % que se inicia neste vértice
    end
    plot(pc(:,1),pc(:,2),'.-g','linewidth', 1.3);
end
clear pc P C  % -> elimina as variáveis usadas nesse bloco
% FIM Parte C
```

```
%{
Título : Aspectos Teóricos e Computacionais da Geração de Malhas Quadtree
Autor  : Fernando Pacanelli Martins (número USP: 5825419)
                  Mestrando em Matemática Aplicada e Computacional
                  Mecânica dos Fluidos Computacional
Data       : 23/11/2007
Sistema    : QUADMESH 2D - gerador de malhas quadtree
Módulo     : RepresentaContornoPeso.m
Descrição  : Instruções para representar o contorno e poligonais de controle
             do domínio, mostrando o nível de refinamento exigido por cada
             ente geométrico.
%}
fid = fopen(ARQFCC,'r');  % -> arquivo de definição do domínio

% Parte A - plotagem do contorno externo
N  = fscanf(fid,'%d', 1); % -> número de vértices do ciclo externo
ce = zeros(N+1,4);        % -> aloca-se um vértice a mais para que se possa
                          % repetir o primeiro vértice na última posição,
                          % de forma a "fechar" o ciclo que define o
                          % contorno externo

for ii = 1:1:N
    ce(ii,1) = fscanf(fid, '%f ',1); % -> abcissa  do vértice
    ce(ii,2) = fscanf(fid, '%f ',1); % -> ordenada do vértice
    ce(ii,3) = fscanf(fid, '%d ',1); % -> nível exigido pelo vértice
    ce(ii,4) = fscanf(fid, '%d ',1); % -> nível exigido pela aresta que se
                                     % inicia neste vértice
end
N = N + 1;          % -> replicação do primeiro vértice na última posição
ce(N,1) = ce(1,1);  % para facilitar a plotagem e fechar o ciclo
ce(N,2) = ce(1,2);
ce(N,3) = ce(1,3);
ce(N,4) = ce(1,4);
plot(ce(:,1),ce(:,2),'.-r');  % -> plotagem do contorno externo
for ii = 1:1:(N-1)
    x = ce(ii,1);
    y = ce(ii,2);
    text(x,y,int2str(ce(ii,3)));  % -> peso do vértice
    x = 0.5*(ce(ii,1) + ce(ii+ 1,1));
    y = 0.5*(ce(ii,2) + ce(ii+ 1,2));
    text(x,y,int2str(ce(ii,4)));  % -> peso da aresta
end
xmin = min(ce(:,1));
ymin = min(ce(:,2));
xmax = max(ce(:,1));
ymax = max(ce(:,2));
lado = max(xmax-xmin,ymax-ymin) + 1; % -> lado do quadrante universo
xmedio = 0.5*(xmin + xmax);
ymedio = 0.5*(ymin + ymax);
axis([xmedio-0.5*lado,xmedio+0.5*lado,ymedio-0.5*lado,ymedio+0.5*lado]);
clear ce N xmin xmax ymin ymax lado xmedio ymedio % -> elimina as variáveis
                                                  % usadas nesse bloco
% FIM Parte A
```

```
% Parte B - plotagem dos contornos internos
I  = fscanf(fid,'%d', 1);    % -> número de ciclos internos
for ii = 1:1:I
    D = fscanf(fid,'%d', 1);  % -> número de vértices em cada ciclo interno
    ci = zeros(D+1,4);
    for jj = 1:1:D
        ci(jj,1) = fscanf(fid, '%f ',1);  % -> abcissa  do vértice
        ci(jj,2) = fscanf(fid, '%f ',1);  % -> ordenada do vértice
        ci(jj,3) = fscanf(fid, '%d ',1);  % -> nível exigido pelo vértice
        ci(jj,4) = fscanf(fid, '%d ',1);  % -> nível exigido pela aresta
                                          % que se inicia neste vértice
    end
    D = D + 1;         % -> replicação do primeiro vértice na última posição
    ci(D,1) = ci(1,1);% para facilitar a plotagem e fechar o ciclo
    ci(D,2) = ci(1,2);
    ci(D,3) = ci(1,3);
    ci(D,4) = ci(1,4);
    plot(ci(:,1),ci(:,2),'.-r');
    for ii = 1:1:(D-1)
        x = ci(ii,1);
        y = ci(ii,2);
        text(x,y,int2str(ci(ii,3)));   % -> peso do vértice
        x = 0.5*(ci(ii,1) + ci(ii+ 1,1));
        y = 0.5*(ci(ii,2) + ci(ii+ 1,2));
        text(x,y,int2str(ci(ii,4)));   % -> peso da aresta
    end
end
clear ci I D   % -> elimina as variáveis usadas nesse bloco
% FIM Parte B


% Parte C - plotagem das poligonais de controle
P  = fscanf(fid,'%d', 1);     % -> número de poligonais de controle
for ii = 1:1:P
    C = fscanf(fid,'%d', 1); % -> número de vértices em cada poligonal
    pc = zeros(C,4);
    for jj = 1:1:C
        pc(jj,1) = fscanf(fid, '%f ',1);  % -> abcissa  do vértice
        pc(jj,2) = fscanf(fid, '%f ',1);  % -> ordenada do vértice
        pc(jj,3) = fscanf(fid, '%d ',1);  % -> nível exigido pelo vértice
        pc(jj,4) = fscanf(fid, '%d ',1);  % -> nível exigido pela aresta
                                          % que se inicia neste vértice
    end
    plot(pc(:,1),pc(:,2),'.-g');
    x = pc(1,1);
    y = pc(1,2);
    text(x,y,int2str(pc(1,3)));  % -> peso do primeiro vértice
    for ii = 2:1:C
        x = 0.5*(pc(ii-1,1) + pc(ii,1));
        y = 0.5*(pc(ii-1,2) + pc(ii,2));
        text(x,y,int2str(pc(ii-1,4)));
        x = pc(ii,1);
        y = pc(ii,2);
        text(x,y,int2str(pc(ii,3)));     % -> peso da aresta
    end
```

```
end
clear pc P C  % -> elimina as variáveis usadas nesse bloco
% FIM Parte C
```

Bibliografia

[BH83] Baehmann, P. L.; "Automated finite element modeling and simulation"; Tese de Doutorado. Rensselaer Polytechnic Institute, Estados Unidos, 1989.

[FT00] Fortuna, A.O.; "Técnicas computacionais para dinâmica dos fluidos: conceitos básicos e aplicações"; EDUSP, 2000.

[FL91] Fletcher, C. A. J.; "Computational techniques for fluid dynamics"; Vol. I, Springer, 1991.

[FS07] de Sousa, F.S.; Notas de Aula: Tópicos em análise numérica - geração de malhas; ICMC, USP, São Paulo, Brasil, 2007.

[LE02] LeVeque,R.J.; "Finite volume methods for hyperbolic problems"; CUP, 2002.

[LF07] Souza, L.F.; Notas de Aula: Introdução à mecânica dos fluidos computacional; ICMC, USP, São Paulo, Brasil, 2007.

[MT94] Micali, J. F. M.; "Modelagem de figuras planas e geração de malhas usando quadtree"; Dissertação de Mestrado. USP, Brasil, 1994.

[QT00] Quarteroni, A.; Sacco, R. ; Saleri, F.; "Numerical mathematics"; Texts in Apllied Mathematics, número 37; Springer, 2000.

[TB90] Tenembaum, A.M.; Langsam, Y.; Augenstein, M. J.; "Data structures using C and C++"; Prentice-Hall, 1996.

[YS83] Yerry, M. A.; Shephard, D. V.; "A modified quadtree approach to finite element mesh generation"; IEEE Comput. Graph & Appl., p. 39-46, frebuary, 1983.

[ZC68] Zienkiewicz, O. C.; Cheung, Y. K.; "The finit element method in structural and continuum mechanics"; McGraw-Hill Publishing Company Limited, 1968.

NOTAS DO ICMC

SÉRIE COMPUTAÇÃO

093/2007 SANTOS, M.O.; TOLEDO, F.M.B.; ARAUJO, S.A. - Uma abordagem utilizando relaxação langrangeana/surrogate para o problema de dimensionamento de lotes e distribuição.

092/2007 LOPES, M.A.L.P.; ARENALES, M.N.; SANTOS, M.O. - Some extensions of dual cuts to one-dimensional cutting stock problems.

091/2007 ALEM JR., D.J.; MUNARI JR, P.A.; ARENALES, M.N.; FERREIRA, P.A.V. - On the cutting stock problem under stochastic demand.

090/2007 CHERRI, A.; ARENALES, M.N.; YANASSE, H. H. - The unidimensional cutting stock problem with usable leftover - a heuristic approach.

089/2007 TOLEDO, F.M.B.; ARMENTANO, V.A. - Branch-and-bound algorithms for capacitated lot-sizing in parallel machines

088/2007 TOLEDO, F.M.B.; SANTOS, M.O.; ARENALES, M.N.; SELEGHIM JR, P. - Logística de distribuição de água em redes urbanas - racionalização energética.

087/2005 GOIS, J.P; ESTÁCIO, K.C.; OISHI, C.M. BERTTONI,V.; BOTTA, V.A.; NAGAMINE, A.; KUROKAWA, F.A.; FEDERSON, F. - Aplicação de volumes finitos na simulação numérica de contaminação em lençóis freáticos.

086/2005 MARQUES, F.P.; ARENALES, M.N. - The constrained compartmentalized knapsack problem.

085/2005 POLDI, K.C.; ARENALES, M.N. - Dealing with small demand in integer cutting stock problems with limited different stock lengths.

084/2005 PRADO, T.A.S.; NUNES, M.G.V. - A statistical generative model for unsupervised learning of web argument structures.