3
Algoritmo proposto para a geração de arranjos

Neste trabalho, é proposto um algoritmo que busca superar algumas dificuldades encontradas pelos algoritmos anteriormente apresentados. Algumas vantagens deste algoritmo são:

- o algoritmo proposto se enquadra no grupo dos algoritmos geométricos, e portanto, tem um baixo custo computacional na geração de arranjos;
- a distribuição granulométrica é definida pelo usuário. As partículas são geradas aleatoriamente seguindo a granulometria prescrita;
- permite geração de um arranjo de partículas denso;
- como as partículas não se sobrepõem, tem-se que, quando do início da análise do MED, as forças iniciais nas partículas serão zero.

O princípio básico do algoritmo é gerar partículas, uma a uma, de modo que elas sejam posicionadas tangencialmente a outros obstáculos — paredes do domínio ou outras partículas anteriormente colocadas — através da busca de interseção de formas geométricas. Um passo-a-passo detalhado do algoritmo será visto a seguir.

O primero passo do algoritmo proposto é a representação do contorno do domínio. Na implementação neste trabalho, o contorno do domínio é representado através da discretização com círculos, como na figura 3.1. Isto não é uma limitação do algoritmo e o contorno pode ser modelado através de retas ou mesmo linhas curvas. Esta escolha se baseia na metodologia adotada no trabalho de Vieira [47] na implementação do MED, para o qual os modelos gerados neste trabalho serão analisados, onde tanto o contorno quanto as partículas são circulares.

O próximo passo é a criação da lista inicial com as possíveis frentes (em inglês, fronts). A frente será melhor abordada mais adiante — quando da sua utilização no algoritmo —, porém, para fins de continuidade, neste passo esta lista é gerada simplesmente ordenando-se as partículas constituintes do contorno segundo uma dada metodologia, de modo a ser garantida uma hierarquia definida. Esta metodologia é arbitrária e foi escolhida de maneira a priorizar as partículas posicionadas mais abaixo e em seguida, como critério

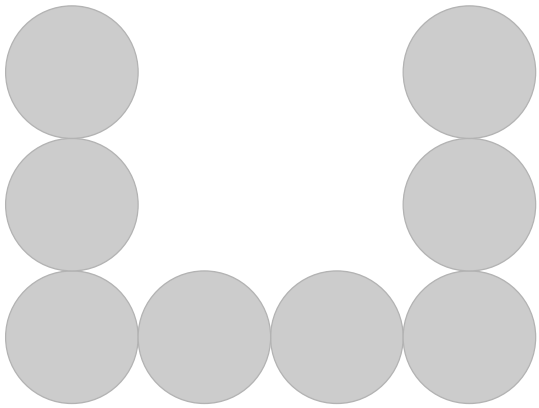

Figura 3.1: Representação do contorno por círculos.

de desempate, aquelas mais à esquerda. Como duas partículas não podem ter um mesmo par de coordenadas, estes critérios são necessários e suficientes para identificar univocamente qualquer partícula em uma análise em duas dimensões.

Em seguida, inicia-se um laço (*loop*) de geração de partículas. Para ser gerada uma partícula, deve-se determinar o tamanho da partícula, além de ter conhecimento das fronteiras do domínio e das partículas anteriormente geradas, de modo a ser possível a geração sem sobreposição. No caso implementado, como as partículas são circulares, o tamanho corresponde ao diâmetro.

Para a geração de cada partícula, inicialmente, obtém-se na lista de possíveis frentes a primeira destas (figura 3.2). Como descrito anteriormente, esta frente nada mais é que uma partícula, e em uma primeira iteração é uma partícula discretizada do contorno. A idéia por trás do algoritmo é gerar partículas que estejam em contato com a frente e com outro obstáculo — a partir deste ponto, como o contorno do domínio é representado por partículas, não será mais chamado obstáculo e sim, partícula.

Depois, para esta frente, procuram-se todas as partículas localizadas em suas proximidades, e por conseqüência, que possam interferir na posição da partícula sendo gerada. A definição de proximidade é relativa, e portanto, neste caso foram arbitradas como estando próximas, teoricamente, todas as partículas que estivessem situadas dentro da área de busca representada por um quadrado com centróide coincidente ao da frente e lado igual a duas vezes a soma do raio da partícula frente, do diâmetro da partícula que está sendo gerada e do raio da maior partícula possível, conforme visto na figura 3.3.

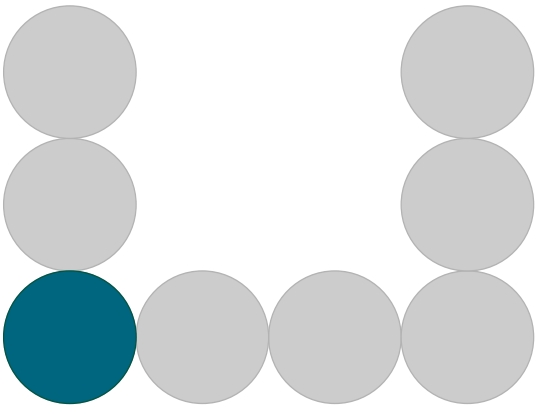

Figura 3.2: Primeira frente — uma das partícula mais abaixo e aquela mais à esquerda.

Na prática, no entanto, um cuidado extra deve ser tomado — o quadrado que descreve a área de busca de proximidade de partículas tem suas bordas superior e direita como intervalo aberto, e por isso, no caso de uma partícula estar localizada sobre uma destas bordas, ela não seria computada para cálculo de interseções. Para resolver tal situação, o lado do quadrado deve ser necessariamente maior do que o calculado anteriormente, porém, quanto maior a área de busca, mais partículas seriam consideradas próximas, aumentando o esforço computacional desnecessariamente. No escopo da implementação deste algoritmo foi arbitrado que ao invés de ser somado o diâmetro da partícula a ser gerado, soma-se este valor incrementado de dez porcento.

Com todas as partículas próximas em mãos, para cada uma delas, assim como para a frente, geram-se o que foram chamados halos (em inglês, também, halos), mostrados na figura 3.4. Halo é um recurso que será utilizado para o cálculo do posicionamento possível do centróide da partícula, e pode ser definido como o lugar geométrico dos pontos cuja distância até qualquer ponto na superfície da partícula é maior ou igual à metade do tamanho da partícula a ser gerada. No caso específico de um círculo de diâmetro d_1, o halo é uma circunferência de diâmetro $d_1 + d_{max}$, onde d_{max} é o tamanho máximo das partículas.

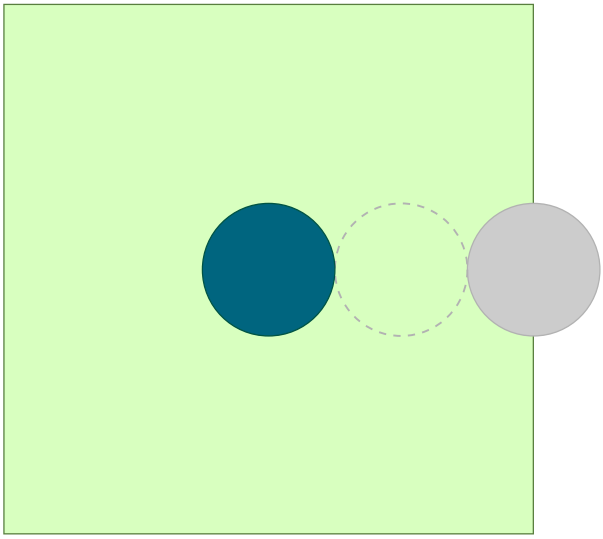

Figura 3.3: Área de busca. A área de busca teórica é representada pelo quadrado verde. O círculo azul representa a frente, o cinza uma partícula com o tamanho máximo e o tracejado seria uma possível partícula a ser gerada. Nota-se que a área de busca engloba toda a área onde pode haver uma partícula que influencie a geração.

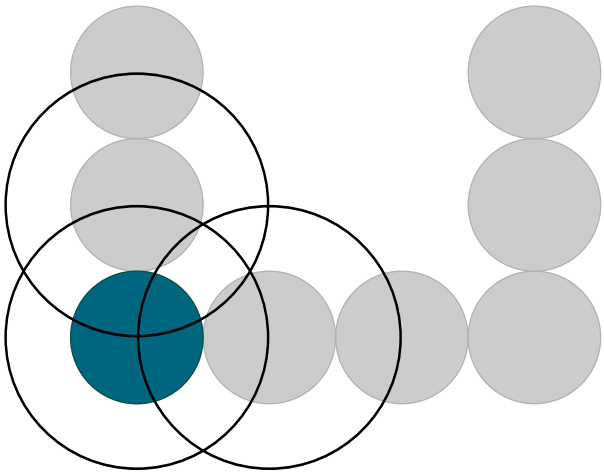

Figura 3.4: Halos das partículas próximas à primeira frente.

O uso do halo se justifica por definir todas as posições candidatas a centróide da partícula a ser gerada. Estas posições determinam-se pelas interseções do halo da frente com cada halo das partículas próximas (figura 3.5). Algumas destas posições, todavia, são impraticáveis, dado que podem estar situadas fora do domínio ou no interior de outros halos (figura 3.6). Em tais circunstâncias, estas posições são descartadas, restando somente as que

foram classificadas no algoritmo como posições praticáveis (figura 3.10).

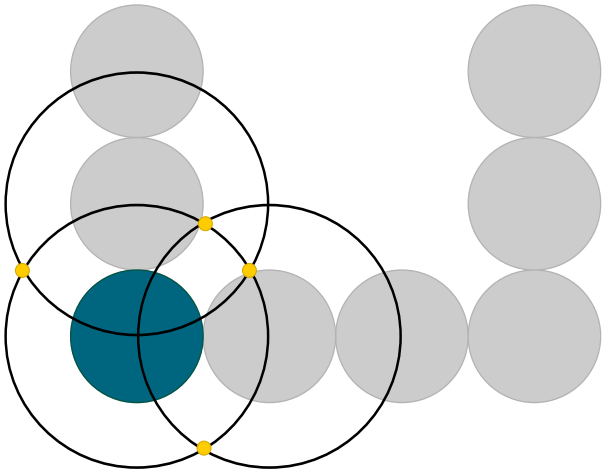

Figura 3.5: Posições candidatas a ser centróide da partícula sendo gerada para a primeira frente.

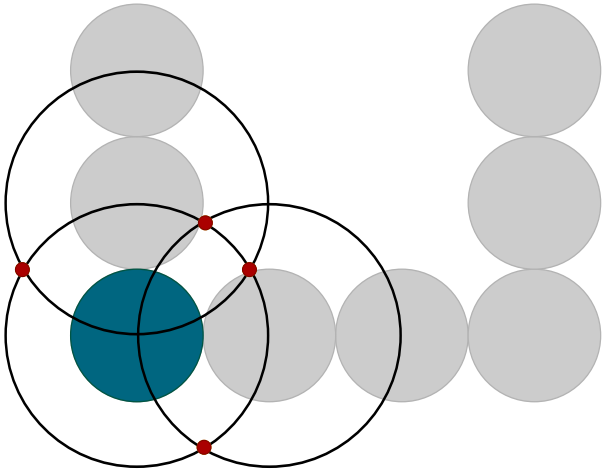

Figura 3.6: Descarte das posições impraticáveis do centróide da partícula a ser gerada para a primeira frente. Não há boas interseções para essa frente.

Há a possibilidade, no entanto, que não se tenha conseguido, para a frente selecionada, encontrar uma interseção sequer onde pudesse ser gerada uma partícula. Neste caso, a frente é descartada da lista de possíveis frentes. Para fins de otimização, se não houver na lista de boas interseções pelo menos duas delas, a frente também é removida de sua lista. A próxima iteração é

iniciada conservando-se, entretanto, o tamanho anterior da partícula (figuras 3.7, 3.8, 3.9 e 3.10).

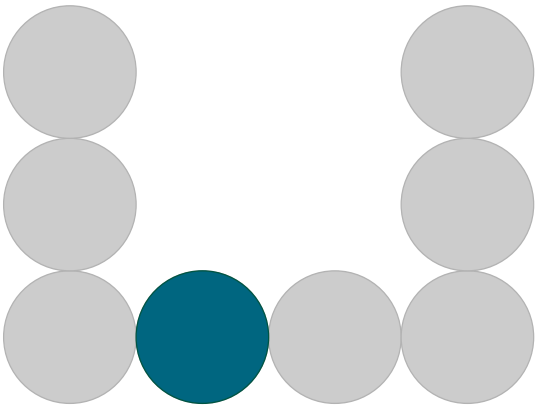

Figura 3.7: Segunda frente — uma das partícula mais abaixo e a segunda mais à esquerda.

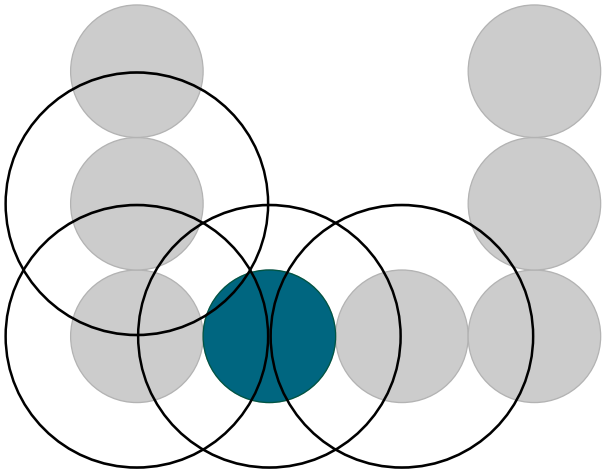

Figura 3.8: Halos das partículas próximas à segunda frente.

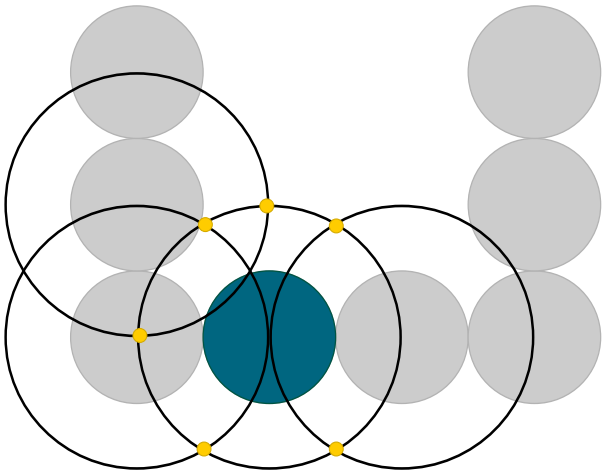

Figura 3.9: Posições candidatas a ser centróide da partícula sendo gerada para a segunda frente.

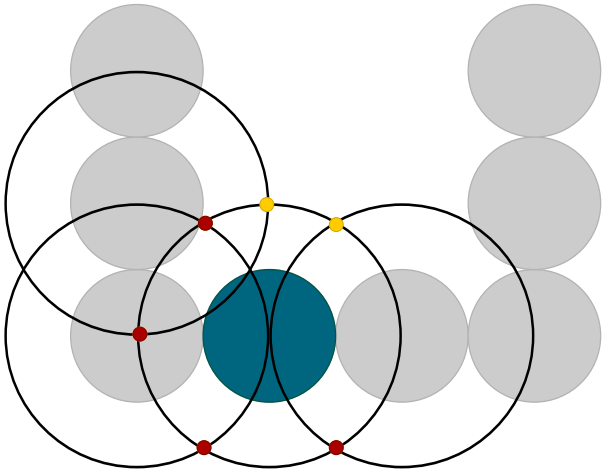

Figura 3.10: Descarte das posições impraticáveis do centróide da partícula a ser gerada para a segunda frente.

Dentre as boas interseções, somente uma pode ser escolhida — e ser designada no algoritmo como melhor interseção (figura 3.11) — e para isso é utilizado o mesmo critério que antes, onde a vantagem é dada à interseção que possua a menor ordenada. Este critério tende a levar a partícula a uma posição de energia potencial mais reduzida, na tentativa de simular mais fidedignamente uma situação de deposição da partícula.

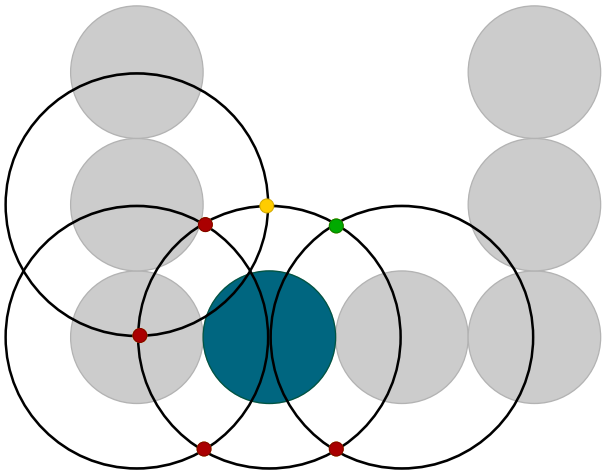

Figura 3.11: Escolha da melhor entre as posições praticáveis do centróide da partícula a ser gerada para a segunda frente.

Ao final, no caso favorável de ser encontrada uma interseção, é gerada uma partícula com o tamanho selecionado e centróide nesta interseção, como apresentado na figura 3.12. A partícula é então colocada no topo da lista das possíveis frentes, e tem começo uma nova iteração, como na figura 3.13.

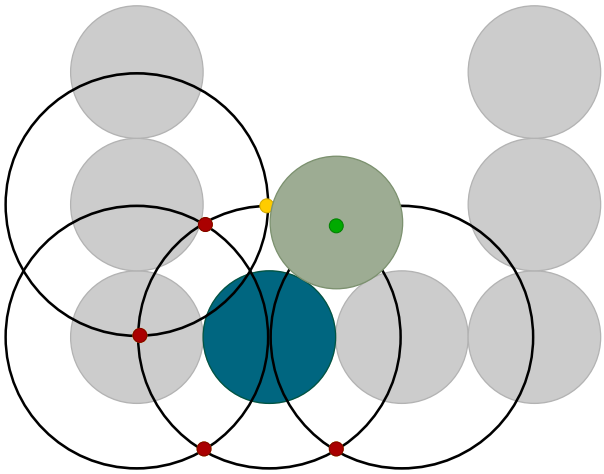

Figura 3.12: Geração de nova partícula na melhor posição do centróide para a segunda frente.

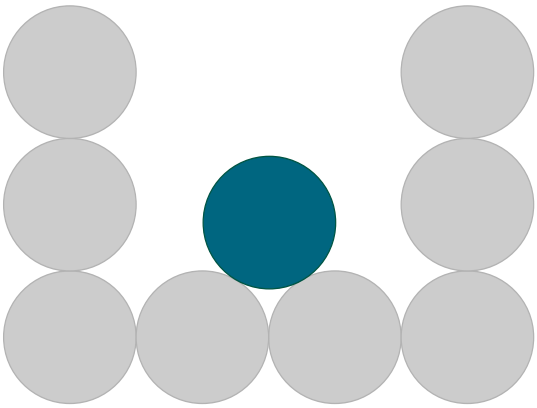

Figura 3.13: Inserção da nova partícula no topo da lista de frentes.

As iterações terminam quando se chega ao final da lista de possíveis frentes ou a lista de tamanhos de partículas. A lista de tamanhos de partículas é definida no início do algoritmo e é constituída por um número suficiente de valores de tamanhos que permitem que o domínio seja preenchido completamente. Para se obter esta lista, é necessário estimar a quantidade máxima de partículas. Esta estimativa foi avaliada com base no domínio e no menor tamanho das partículas, segundo a fórmula 3-1.

$$n_{est} = \left\lceil \frac{a_c}{a_{p,min}} \right\rceil \tag{3-1}$$

Onde,

n_{est} quantidade estimada de partículas

a_c área limitada pelo domínio

$a_{p,min}$ área da partícula mínima

3.1 Otimização do desempenho do algoritmo

Uma preocupação com relação ao algoritmo é que a busca das partículas próximas à frente pode ser extremamente custosa computacionalmente, principalmente tendo-se em vista que será gerado um número considerável de partículas e somente poucas delas podem estar efetivamente em contato com a frente. Para avaliar o desempenho relativo desta busca, foi executado um

benchmark na implementação do algoritmo, de maneira que pudesse ser confirmado que o gargalo da aplicação seria a busca. O domínio é quadrado de lado 100mm e a granulometria é definida pela tabela 3.1. A tabela 3.2 apresenta o resultado do *benchmark*, mostrando somente os dez métodos mais custosos computacionalmente, em porcentagem.

Tabela 3.1: Granulometria utilizada no *benchmark*.

Diâmetro(mm)	Passante acumulado(%)
1,275	0
1,800	35
2,550	95
3,540	100

Tabela 3.2: *Benchmark* da implementação inicial do algoritmo.

Método	Tempo		Invocações
Todos	49.796 ms	(100,0%)	
ListNode.getPoints()	16.693 ms	(33,5%)	10.317
LeafNode.getPoints()	5.403 ms	(10,9%)	10.915.714
query(Rectangle2D)	1.847 ms	(3,7%)	10.315
containsPoint(Particle, Point2D)	1.132 ms	(2,3%)	1.142.197
getGoodIntersections(List, List)	1.017 ms	(2,0%)	5.158
containsPoint(CircularParticle, Point2D)	733 ms	(1,5%)	1.142.197
intersections(CircularParticle, ...)	192 ms	(0,4%)	107.078
makeHalo(double)	112 ms	(0,2%)	112.236
intersections(Particle, Particle)	108 ms	(0,2%)	107.078
getIntersections(Particle, List)	101 ms	(0,2%)	5.158

Pode-se notar que os dois primeiros métodos são responsáveis por 44,4% do esforço computacional da implementação do algoritmo e portanto uma ligeira explicação destes se torna necessária, de modo a dar prosseguimento ao texto.

Os método `ListNode.getPoints()` e `LeafNode.getPoints()` são chamados pelo método `rangeSearch(Rectangle2D)`, que por sua vez, é chamado pelo método `getNearParticles(Particle, double)`. Este método é responsável por retornar as partículas que estariam próximas a uma dada partícula e como esperado, o gargalo é a busca das partículas. Uma possível maneira de melhorar o desempenho do algoritmo de busca por partículas próximas é utilizar uma estrutura de dados, como é explicado a seguir.

Estruturas de dados são um modo de armazenar e organizar dados de forma a poderem ser utilizados eficientemente por um computador. Diferentes estruturas de dados são adequadas a diferentes problemas e podem ser ex-

tremamente especializadas. Para problemas de particionamento de um espaço bidimensional, *quadtrees* se mostram uma estrutura de dados interessante [40].

3.1.1 Quadtrees

As *quadtrees* são árvores baseadas na decomposição recursiva do espaço bidimensional em quatro regiões — normalmente chamados quadrantes nordeste, sudeste, sudoeste e noroeste —, daí sua denominação. Seu análogo em três dimensões é a *octree*. Diversos tipos de *quadtree* foram propostas, cada uma adaptada a diferentes tipos de busca em domínios espaciais. Um exemplo natural em um espaço bidimensional é a busca por latitudes e longitudes de cidades, entretanto, qualquer procura em um banco de dados que contenha dois parâmetros também pode ser considerado assim, como uma busca por pessoas entre um certo intervalo de idade e de renda, por exemplo. Por ser possível representar pares de parâmetros como coordenadas de pontos em um gráfico, eles serão referidos como pontos de dado.

Embora os vários tipos de *quadtree* compartilhem algumas características básicas, algumas diferenças podem ser notadas. Segundo Sperber [44], existem alguns tipos básicos de *quadtrees*, que serão brevemente explicadas.

Point quadtree

Point quadtree é uma combinação de grade uniforme e árvore binária proposta por Finkel e Bentley [19] (figura 3.14, retirada de Samet [40]).

O método de construção é colocar o primeiro ponto de dado no nó da raiz e depois subdividir o espaço em quadrantes, tendo o ponto como centro da subdivisão. As inserções subseqüentes são recursivamente aplicadas através da descoberta do quadrante no qual o ponto está contido e, no caso de estar vazio, da adição do ponto e seguinte subdivisão em outros quadrantes, como anteriormente feito. Se, por outro lado, o quadrante não estiver vazio, procura-se o subquadrante onde o ponto está situado.

Do mesmo modo que a árvore binária, a estrutura do *point quadtree* depende da ordem dos pontos inseridos.

Há alguns algoritmos para deleção, mas eles não são considerados ideais.

A *point quadtree* é especialmente indicada para busca de vizinho mais próximo (em inglês, *nearest neighbour search*).

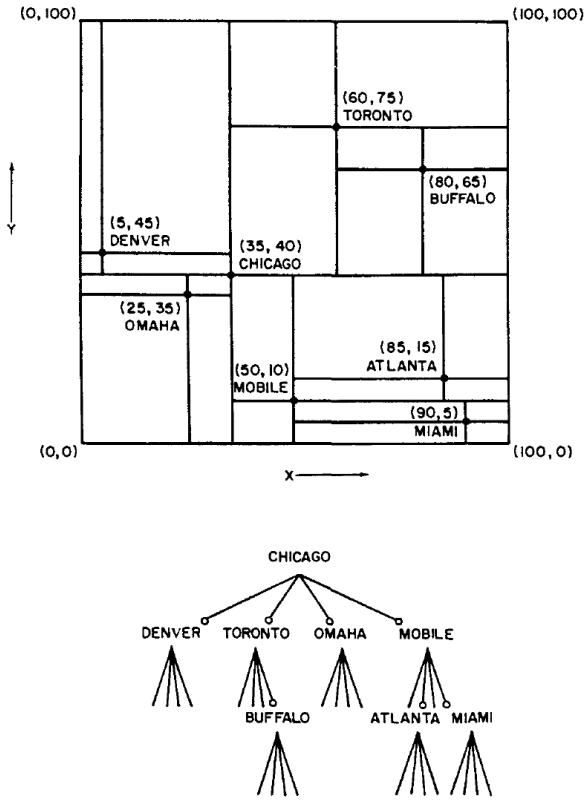

Figura 3.14: *Point quadtree* e os dados que ela representa.

MX quadtree

Outro tipo de *quadtree* são as chamadas *region quadtrees*. As *region quadtrees* têm por característica dividir o espaço regularmente. Há muitas versões de *region quadtree*, uma delas chamada *MX quadtree*. A figura 3.15, modificada de figura retirada de Samet [40], mostra o processo de inclusão de um primeiro ponto na *quadtree*. A figura 3.16, retirada de Samet [40], mostra uma *MX quadtree* com vários pontos incluídos.

A *MX quadtree* decompõe um espaço finito em regiões quadradas de dimensões um por um, como uma matriz — MX vem de *matrix*, em inglês — e seus nós-folha indicam se existe ou não um ponto de dado naquela posição na matriz.

Os pontos na *MX quadtree* são envoltos por um quadrado de lado 2^N, onde N é inteiro, dado que cada subdivisão da região ocorre no ponto central. Se for necessário representar uma região que não seja quadrada, deve ser criada uma *MX quadtree* que seja grande o suficiente para envolver toda a região.

Diferentemente da *point quadtree*, somente os nós-folhas podem armazenar dados.

Essas árvores são bem eficientes em termos de memória no caso de representação de matrizes esparsas. É uma boa opção para busca de k-vizinhos

mais próximos (em inglês, *k-nearest neighbors search*).

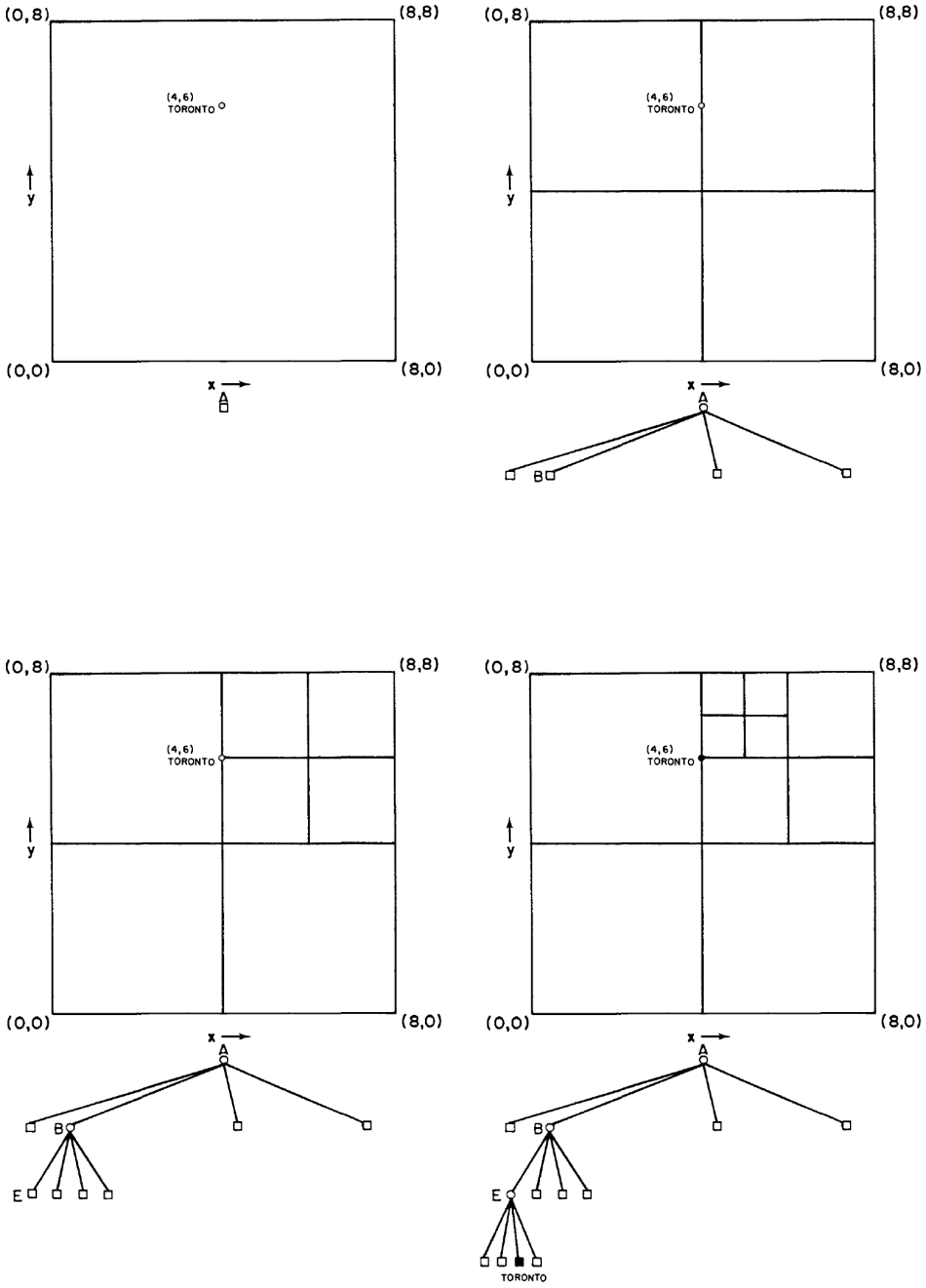

Figura 3.15: Processo de inclusão de um ponto na *MX quadtree*.

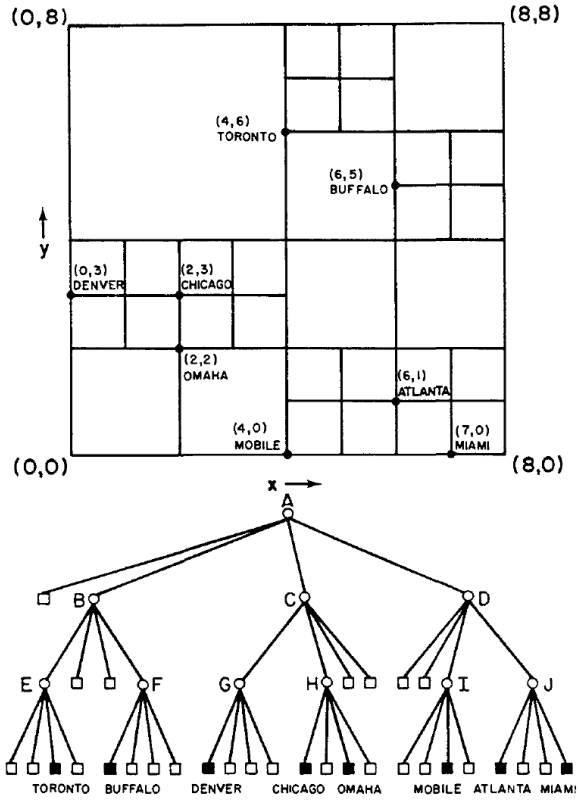

Figura 3.16: *MX quadtree* e os dados que ela representa.

PR quadtree

Outra *region quadtree* é a *PR quadtree* (figura 3.17, retirada de Samet [40]), cuja diferença em relação à *MX quadtree* é que a subdivisão em quadrantes é baseada nos pontos inseridos. Os pontos são inseridos como numa *point quadtree*, mas, como numa *MX quadtree*, os dados são inseridos nos nós-folha — PR abrevia *point region*. Na inserção de pontos de dados, ao ser alcançado um nó-folha que não esteja vazio, o espaço precisa ser subdividido até que somente um ponto esteja contido por quadrante. Isto torna a inserção mais complicada que em uma *point quadtree* ou em uma *MX quadtree*.

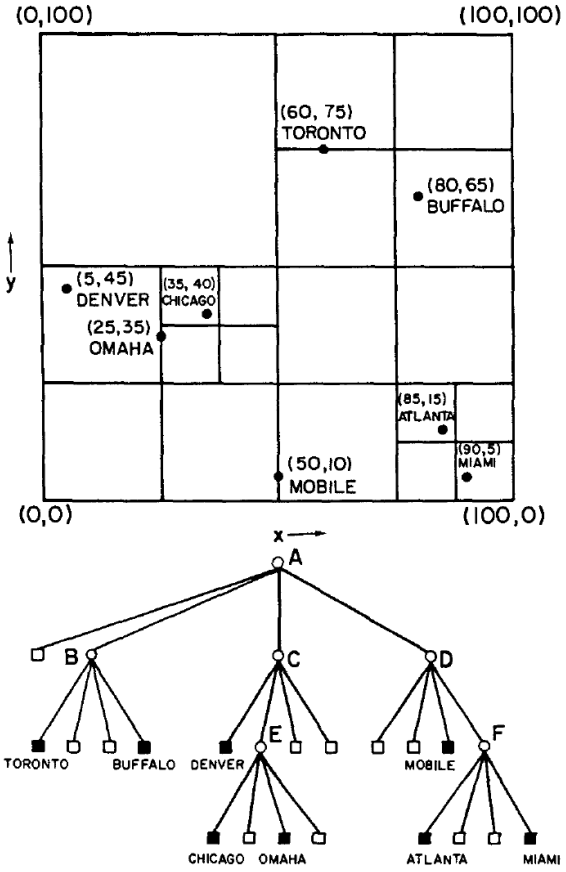

Figura 3.17: *PR quadtree* e os dados que ela representa.

Bucket PR quadtree

A *bucket PR quadtree* é basicamente uma *PR quadtree*. A diferença é que o espaço é subdividido somente quando o número de pontos de dados em cada nó extrapola a capacidade do balde (*bucket*), como visto na figura 3.18 para uma capacidade do balde igual a 2 (retirada de Samet [40]).

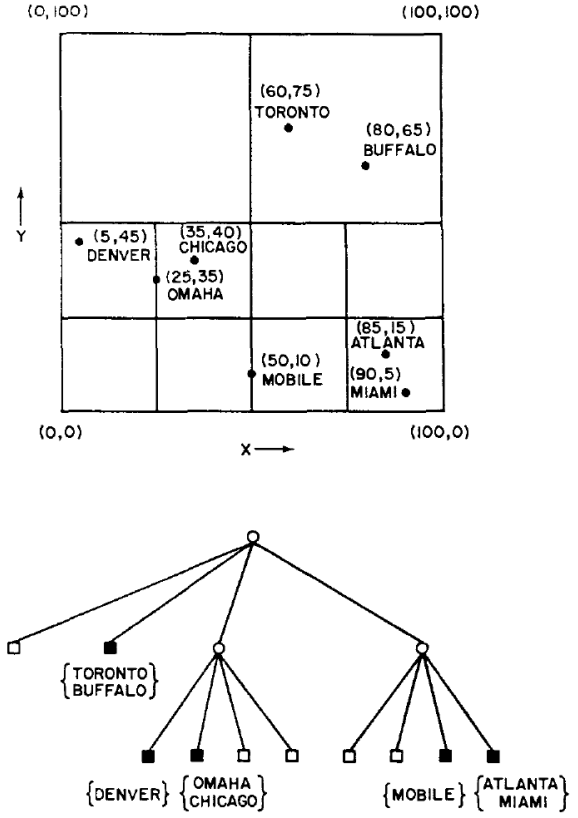

Figura 3.18: *Bucket PR quadtree* para uma capacidade do balde igual a 2 e os dados que ela representa.

3.1.2 Escolha da quadtree

Como o problema abordado na implementação do algoritmo é a busca pelas partículas que estejam próximas à frente e o espaço a ser preenchido é finito, a escolha da *MX quadtree* é natural. Seu desempenho na implementação do algoritmo foi avaliado e os dez métodos mais custosos computacionalmente são mostrados na tabela 3.3, para os mesmos parâmetros da avaliação anterior.

Ao serem comparados os *benchmarks* relativos às duas implementações, a original e aquela que utilizou a *MX quadtree*, notamos uma melhora significativa no desempenho — uma redução de 67,3% no tempo computacional —, e os métodos críticos na avaliação anterior foram mitigados: o método `ListNode.getPoints()`, que possui como contrapartida na segunda implementação o método `QuadtreeNode.getPoints()`, que tomava 16.693ms (33,5%), passou a 185ms (1,1%), enquanto o método `LeafNode.getPoints()`, o segundo mais custoso na implementação original com 10,9% e que tomava 5.403ms, não mais figura sequer entre os dez métodos mais custosos na segunda implementação, com menos de 0,8% do total — para constar, toma 56ms, cerca de 0,3%.

Tabela 3.3: *Benchmark* da implementação do algoritmo utilizando a *MX quadtree*.

Método	Tempo		Invocações
Todos	16.287 ms	(100,0%)	
query(Rectangle2D, Node, Rectangle2D)	1.875 ms	(11,5%)	225.742
containsPoint(Particle, Point2D)	1.380 ms	(8,5%)	1.280.257
getGoodIntersections(List, List)	1.074 ms	(6,6%)	5.128
containsPoint(CircularParticle, Point2D)	727 ms	(4,5%)	1.280.257
getQuadrantBounds(Rectangle2D, Quadrant)	338 ms	(2,1%)	732.169
getSon(Quadrant)	267 ms	(1,6%)	761.917
Quadrant.values()	212 ms	(1,3%)	186.872
intersections(CircularParticle, ...)	205 ms	(1,3%)	128.168
QuadtreeNode.getPoints()	185 ms	(1,1%)	77.360
makeHalo(double)	133 ms	(0,8%)	133.326

Algo que deve ser ponderado na análise dos *benchmarks* é que o tempo registrado não pode ser levado em consideração em termos absolutos, já que há uma sobrecarga computacional gerada pelos próprios *benchmarks*. Para fins de medição relativa de custo computacional entre os métodos, no entanto, esses valores são perfeitamente válidos. Nesse sentido, nota-se claramente que o objetivo da redução do trabalho computacional foi cumprido com a utilização da estrutura de dados escolhida, mostrando-se uma decisão acertada.

3.2 Validação do algoritmo

O processo de validação dos resultados ocorreu em duas partes: primeiramente foi conduzida uma comparação com o programa PFC2D e a seguir, uma série de testes foi realizada para por à prova o desempenho e a escalabilidade do algoritmo.

3.2.1 Comparação com algoritmos utilizados no programa PFC2D

Para efeito de comparação, está sendo reproduzida a geração de arranjo através do programa PFC2D como vista em Huamán [21]. No trabalho supracitado foram estudados quatro cenários de preenchimento de fraturas com partículas de sustentação. Neste trabalho, só irá ser reproduzido o cenário 1, pois esse foi o único a ser modelado em duas dimensões.

O domínio tem por dimensões largura e comprimento normalizado dados, respectivamente, pelas fórmulas $W = (W_r + 2) \cdot d_p$ e $L_r = L/d_p$, onde a largura normalizada W_r é igual a 4,88 e o diâmetro médio d_p, 1,25mm. O comprimento normalizado adotado foi igual a 50, *apud* Asgian et al. [6],

cujo trabalho concluiu, após inúmeras simulações numéricas, que este valor representa adequadamente um modelo para estudo do refluxo de propantes utilizando o MED.

Conseqüentemente, as dimensões finais a serem utilizadas nesta comparação são largura igual a 8,6mm e comprimento igual a 62,5mm.

A granulometria segue a distribuição dada pela tabela 3.4 e é mostrada na figura 3.19.

Tabela 3.4: Dados granulométricos utilizados.

Diâmetro(mm)	Passante acumulado(%)
1,200	0
1,215	11
1,225	21
1,235	31
1,245	43
1,255	54
1,265	65
1,275	77
1,285	89
1,295	100

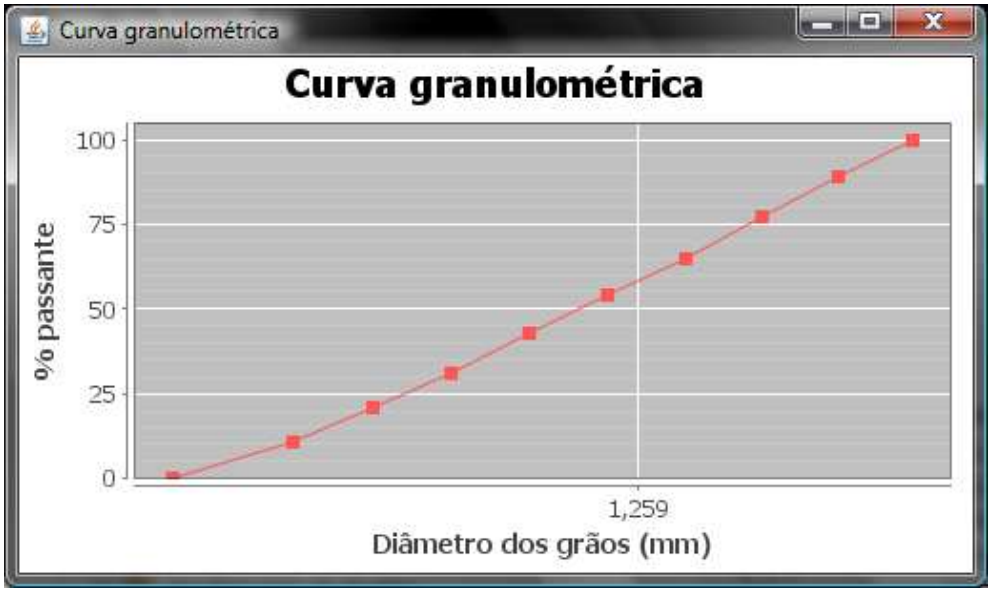

Figura 3.19: Curva granulométrica utilizada.

Para a geração do arranjo através do programa PFC2D, utiliza-se uma linguagem de programação que se encontra embutida nele chamada FISH. Esta linguagem permite ao usuário definir novas variáveis e funções, extendendo assim o programa PFC2D [1].

No PFC2D não existe uma maneira generalizada de geração de arranjos e por isso, alguns métodos serão abordados: a geração por tentativas, por expansão do raio e por repulsão explosiva.

Geração através do algoritmo proposto

Para a geração através do algoritmo proposto, são necessários somente como parâmetros o domínio e a granulometria. Os resultados podem ser vistos na tabela 3.5 e um exemplo é mostrado na figura 3.20.

Tabela 3.5: Resultado da geração através do algoritmo proposto.

Tempo(ms)	Partículas geradas	Porosidade
234	339	0,225
46	338	0,227
47	339	0,224

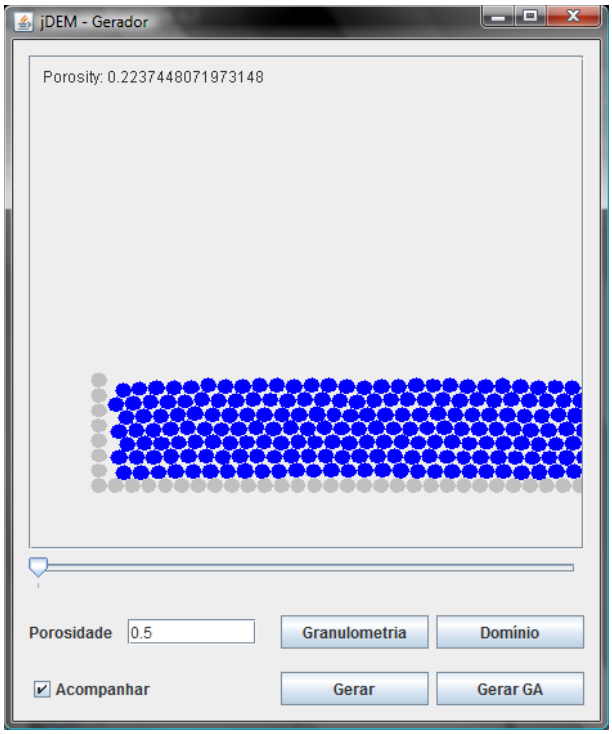

Figura 3.20: Geração através do algoritmo proposto.

Embora o tempo da primeira geração seja muito superior ao das subseqüentes, não há erro nestes valores. Essa discrepância foi produzida pelo compilador *just-in-time* (JIT) do Java.

O JIT converte, em tempo de execução, as instruções de *bytecode* para código de máquina, incrementando o desempenho. O código de máquina resultante da compilação JIT é armazenado na memória, garantindo que o trecho de código em questão não será mais recompilado ou reinterpretado

sempre que, durante a execução do programa, for novamente acionado. Além disso, muitos compiladores JIT possuem mecanismos para realizar otimizações adicionais nos trechos de código do programa que são executados com maior freqüência.

De forma a obter resultados mais homogêneos, antes de cada execução, o programa foi recompilado. Desta forma, simula-se o ato de executar o programa pela primeira vez. Fica óbvio que, no caso de um uso mais freqüente, a velocidade seria aumentada. Estes segundos resultados podem ser vistos na tabela 3.6.

Tabela 3.6: Resultado da geração através do algoritmo proposto, sem otimização.

Tempo(ms)	Partículas geradas	Porosidade
218	339	0,225
218	337	0,224
219	338	0,225

Nos métodos a seguir, é passado como parâmetro a porosidade. Para efeito de comparação, dado que o algoritmo proposto neste trabalho não prescreve uma porosidade, um valor médio obtido do arranjo acima é adotado, igual a $0,225$.

Geração por tentativas no programa PFC2D

O primeiro método de geração testado no PFC2D foi o por tentativas. Como o nome implica, esse método consiste em gerar um arranjo colocando partículas aleatoriamente no domínio, sem que haja sobreposição. Como é possível imaginar, quanto mais cheio estiver o domínio, mais difícil será colocar outra partícula.

Nesse método, são necessários como parâmetros: o domínio, a granulometria e o número máximo de partículas n_{max} que tentarão ser colocadas. Ressalta-se aqui que, por imposição do programa PFC2D, a granulometria foi estipulada uniformemente distribuída entre $1,200$ e $1,295$. Os resultados podem ser vistos na tabela 3.7 para $n_{max} = 20.000$ — que é o padrão no programa — e um exemplo é mostrado na figura 3.21.

Pode-se notar que a porosidade do arranjo é muito elevada, e no intuito de tentar diminuí-la, gerou-se novamente arranjos, só que desta vez, utilizando $n_{max} = 200.000$ com os resultados obtidos mostrados na tabela 3.8, e $n_{max} = 2.000.000$, na tabela 3.9.

Uma comparação destes resultados pode ser vista no gráfico 3.22. Pode-se notar um aumento muito grande no tempo computacional para um ganho

Tabela 3.7: Resultado da geração por tentativas no programa PFC2D — 20.000 tentativas.

Tempo(ms)	Partículas geradas	Porosidade
30	213	0,516
50	210	0,523
30	203	0,537

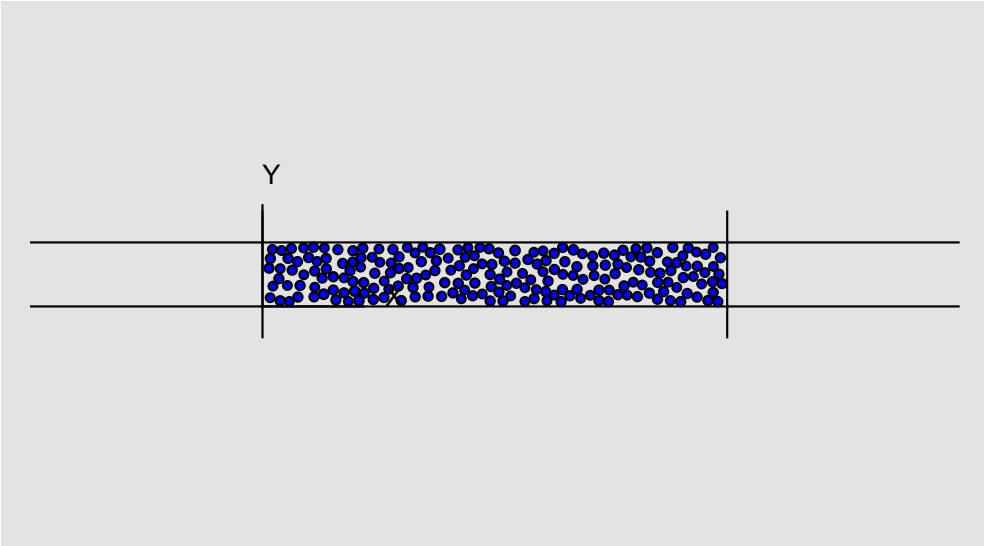

Figura 3.21: Resultado da geração por tentativas no programa PFC2D.

Tabela 3.8: Resultado da geração por tentativas no programa PFC2D — 200.000 tentativas.

Tempo(ms)	Partículas geradas	Porosidade
200	217	0,509
210	221	0,497
200	214	0,513

Tabela 3.9: Resultado da geração por tentativas no programa PFC2D — 2.000.000 tentativas.

Tempo(ms)	Partículas geradas	Porosidade
1820	231	0,474
1810	229	0,479
1820	227	0,482

muito reduzido na porosidade, como era esperado. Este método não é recomendado nem mesmo pelo manual do PFC2D [1], sendo aconselhado escrever uma função em FISH para automatizar o processo que assegure que a porosidade desejada seja obtida .

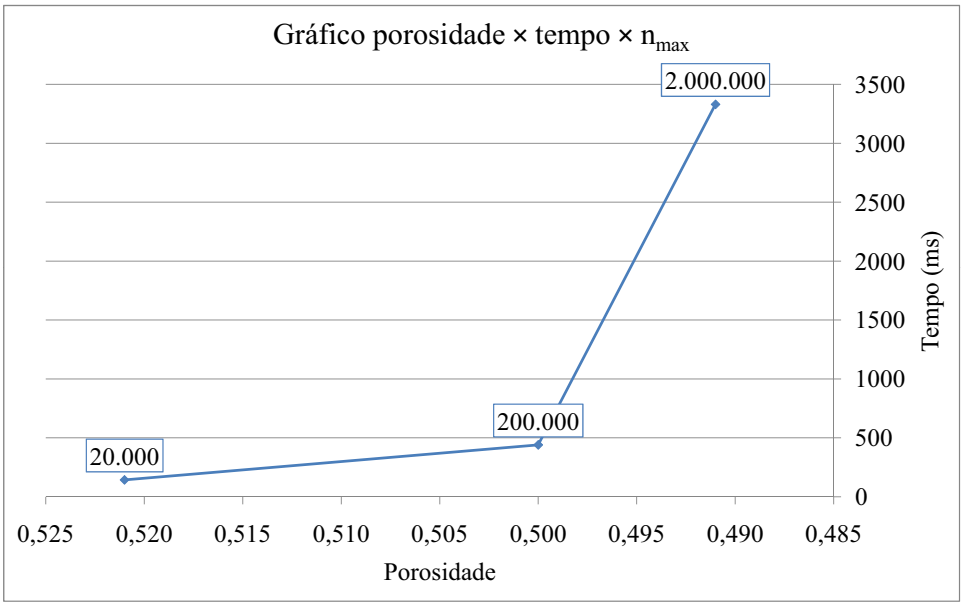

Figura 3.22: Comparação da geração por tentativas no programa PFC2D para diferentes n_{max}.

Geração por expansão do raio no programa PFC2D

No método anterior, não era possível gerar um arranjo com uma dada porosidade, e pior, a porosidade alcançada não é razoável para a maioria das simulações no MED. Este próximo método — geração por expansão de raio — atinge uma porosidade prescrita através da criação das partículas com raios menores e subseqüente expansão gradual desses raios, buscando o equilíbrio do arranjo.

Os parâmetros necessários são: o domínio, a porosidade, o número de partículas n e a razão r entre o raio da maior partícula e da menor. A razão foi naturalmente estabelecida — baseada na granulometria — como $r = 1,295/1,200$, porém o número de partículas traz um desafio — é preciso estimar quantas partículas são precisas para se chegar àquela dada porosidade, tendo-se em vista uma granulometria desejada. Uma opção é calcular um número de partículas aproximado baseado nos dados de entrada. Neste caso tem-se:

$$d_{med} = \frac{1,200 + 1,295}{2} \approx 1,250$$

$$A_{p,med} = \pi \cdot \frac{d_{med}^2}{4} = 1,23$$

$$A_d = w \cdot l = 8,6 \cdot 62,5 = 537,5$$

$$n_{aprox} = A_d / A_{p,med} \approx 438$$

O resultado pode ser visto na tabela 3.10.

Tabela 3.10: Resultado da geração por expansão do raio no programa PFC2D — 438 partículas.

Tempo(ms)	Diâmetro mínimo (mm)	Diâmetro máximo (mm)
520	1,058	1,142
580	1,057	1,142
700	1,058	1,142

Para fins de comparação, como um número de partícula médio pode ser inferido do resultado da geração através do algoritmo proposto — tabelas 3.5 e 3.6 —, este valor será utilizado em outro teste, sendo igual a 338.

Os resultados podem ser vistos na tabela 3.11 e um exemplo é mostrado na figura 3.23.

Tabela 3.11: Resultado da geração por expansão do raio no programa PFC2D — 338 partículas.

Tempo(ms)	Diâmetro mínimo (mm)	Diâmetro máximo (mm)
510	1,205	1,300
520	1,202	1,297
440	1,205	1,301

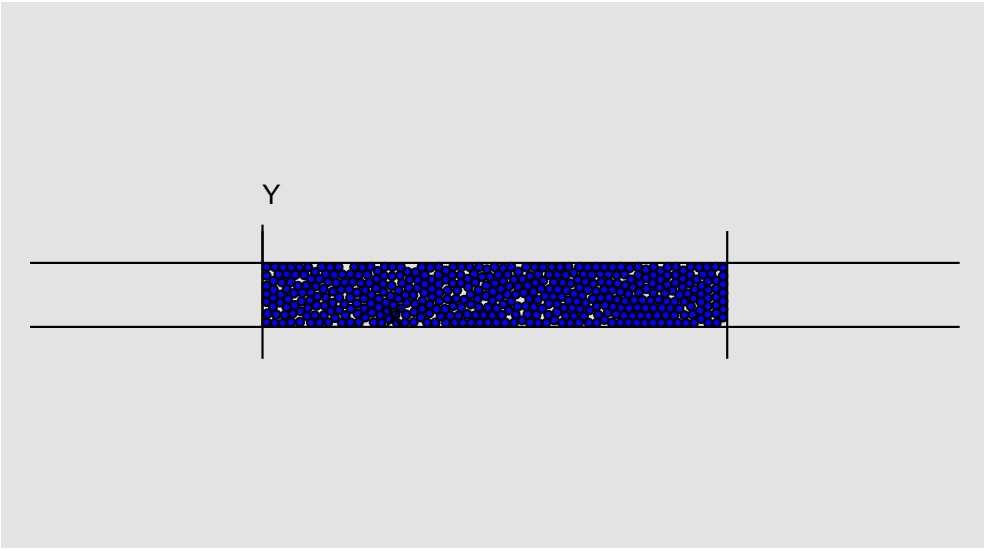

Figura 3.23: Resultado da geração por expansão do raio no programa PFC2D — 338 partículas.

Geração por repulsão explosiva no programa PFC2D

No método de geração por repulsão explosiva, as partículas são criadas aleatoriamente no domínio, já com seus raios finais, e em número suficiente para atingir a porosidade desejada. A sobreposição é permitida, e no caso de ser grande, assim também serão as forças. Tais forças podem gerar velocidades iniciais suficientemente altas, o que permite que algumas partículas escapem através das paredes do domínio. Para prevenir este tipo de acontecimento, durante os primeiros ciclos da convergência para o equilíbrio, várias vezes a energia cinética é reduzida a zero.

Os parâmetros necessários para este método são: o domínio, a granulometria, a porosidade e o número máximo de partículas. Este número máximo é necessário para prevenir, no caso de erro nos dados, que muitas partículas sejam geradas. Este número máximo foi dado igual a 1000.

Os resultados podem ser vistos na tabela 3.12 e um exemplo, na figura 3.24.

Tabela 3.12: Resultado da geração por repulsão explosiva no programa PFC2D.

Tempo(ms)	Partículas geradas	Porosidade
420	340	0,225
430	341	0,226
420	340	0,226

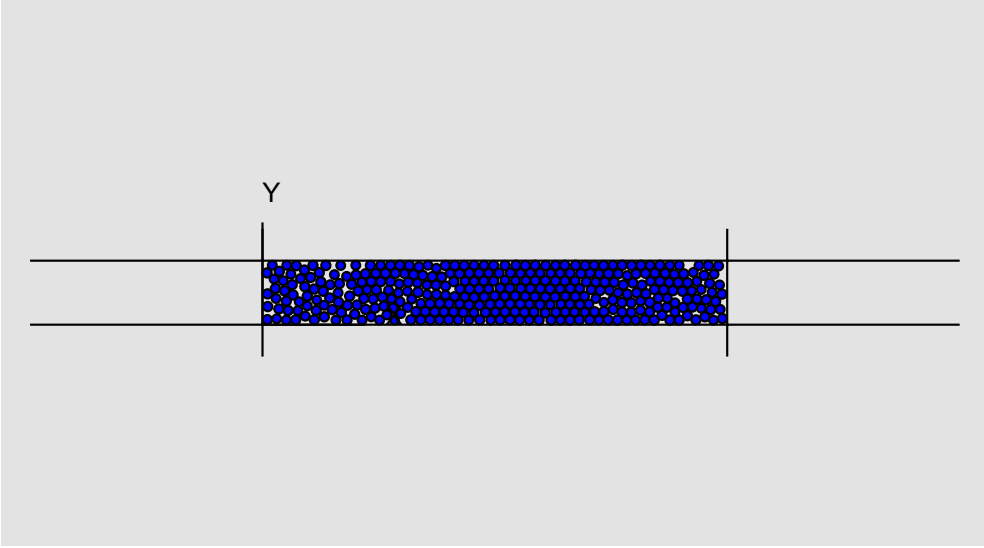

Figura 3.24: Resultado da geração por repulsão explosiva no programa PFC2D.

Considerações sobre a comparação

É importante que se faça algumas considerações sobre os três métodos de geração do PFC2D em comparação com o algoritmo de geração proposto:

- com o algoritmo de geração por tentativas, a porosidade obtida é muito elevada;
- com o algoritmo de geração por expansão do raio, não é possível, efetivamente, prescrever uma granulometria;
- com o algoritmo de geração por repulsão explosiva, embora do ponto de vista puramente de geração seja o melhor dos três métodos, pois gera um arranjo com uma dada porosidade e uma dada granulometria, esse arranjo pode ser extremamente não uniforme em relação à porosidade local, além de possuir grandes forças internas [1].

Para o caso estudado — que possui dimensões modestas—, ambos os programas — o desenvolvido neste projeto e o PFC2D — geram arranjos com rapidez, entretanto, o PFC2D necessita de uma maior interação com o usuário, incluindo a difícil conjectura de alguns parâmetros. Além disso, dependendo do método, alguns efeitos colaterais que devem ser levados em consideração podem surgir.

3.2.2 Investigação do desempenho

De modo a verificar genericamente o desempenho do algoritmo, foi testada uma série de configurações crescente de domínios quadrados. A granulometria segue a distribuição dada pela tabela 3.4 e é mostrada na figura 3.19. Os resultados são apresentados na tabela 3.13 e no gráfico 3.25.

Para testar a implementação do algoritmo em relação a um grande número de partículas, naturalmente é necessário aumentar a razão entre a área do domínio e a área média das partículas. Os testes aqui executados foram com grandes dimensões. Os resultados são mostrados na tabela 3.14.

Um teste em condições reais se faz necessário também, de maneira que não se perca o objetivo real de um algoritmo desse tipo. Para isso, dados de fraturas de poços reais da Petrobras foram obtidos do trabalho de Cachay [9]. A geometria do modelo de fratura adotada neste trabalho é apresentado na figura 3.26. Os dados de fraturas dos poços escolhidos para a análise e os resultados podem ser vistos nas tabelas 3.15 e 3.16, respectivamente.

Tabela 3.13: Resultado da geração de partículas em domínios quadrados.

Dimensões (mm × mm)	Tempo(ms)	Partículas geradas	Porosidade
10 × 10	109	54	0,340
	109	54	0,336
	110	53	0,346
12 × 12	110	80	0,322
	141	80	0,322
	125	81	0,313
15 × 15	156	135	0,264
	172	132	0,279
	156	135	0,264
20 × 20	234	243	0,252
	218	249	0,235
	204	245	0,245
50 × 50	407	1.624	0,202
	312	1.634	0,196
	313	1.616	0,207
100 × 100	640	6.610	0,187
	609	6.566	0,193
	641	6.675	0,180
120 × 120	797	9.542	0,186
	766	9.514	0,188
	812	9.558	0,184
150 × 150	1.141	14.810	0,191
	1.110	14.898	0,186
	1.110	14.905	0,186
200 × 200	1.656	26.546	0,184
	1.688	26.556	0,184
	1.734	26.593	0,183
500 × 500	9.344	166.642	0,181
	9.469	166.875	0,180
	9.578	169.416	0,167
1000 × 1000	37.563	674.027	0,171
	37.078	668.268	0,179
	37.235	667.820	0,179

3.3 Exemplos suplementares

São apresentados agora alguns exemplos que demonstram as possibilidades de aplicação da implementação do algoritmo proposto em domínios de formatos diversos.

As figuras 3.27 e 3.28 apresentam a geração de arranjos em um domínio trapezoidal de altura 20mm, e bases 10mm e 20mm. A primeira utiliza granulometria constante igual a 1,2mm e atinge uma porosidade de 0,233, enquanto a segunda utiliza granulometria uniformemente distribuída entre 1,2 e 2,4 e atinge uma porosidade de 0,283.

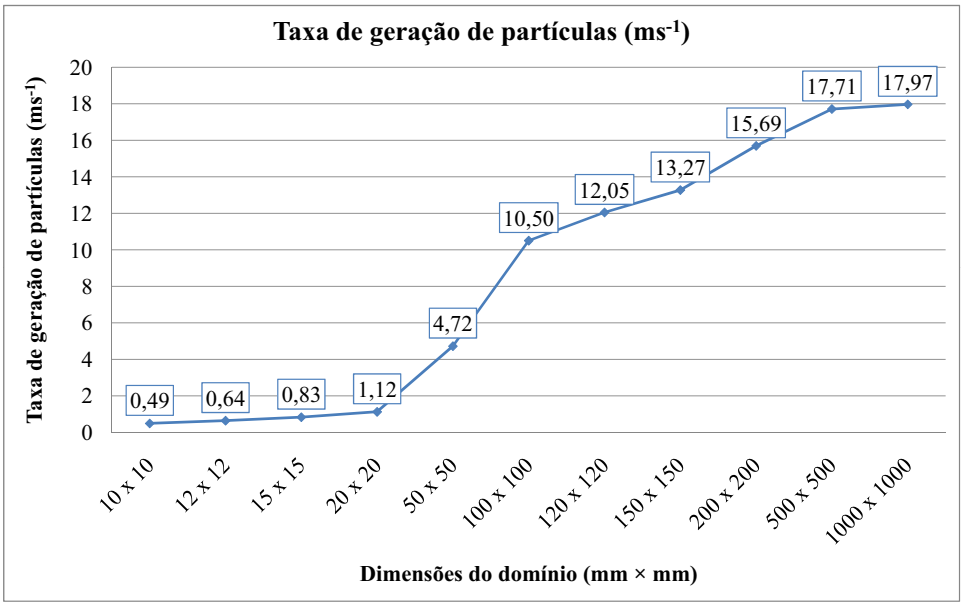

Figura 3.25: Taxa de geração de partículas.

Tabela 3.14: Resultado da geração de partículas em grandes dimensões.

Dimensões (mm × m)	Tempo(ms)	Partículas geradas	Porosidade
10 × 100	40.516	640.684	0,213
	40.860	640.668	0,213
	40.531	640.691	0,213
12 × 100	52.250	821.904	0,158
	52.078	783.259	0,198
	52.562	790.121	0,191
15 × 100	63.187	961.862	0,212
	62.484	961.868	0,212
	63.094	961.907	0,212

Tabela 3.15: Dados de fraturas dos poços da Petrobras.

Poço	Largura (mm)	Altura (m)
7–CP–0370–SE	2,6	8,0
7–CP–1414–SE	2,3	25,0
7–CP–0037–SE	4,2	17,0

As figuras 3.29 e 3.30 apresentam a geração de arranjos em um domínio trapezoidal invertido em relação ao anterior, mais ainda de altura 20mm, e bases 10mm e 20mm. A primeira utiliza granulometria constante igual a 1,2mm e atinge uma porosidade de 0,233, enquanto a segunda utiliza granulometria uniformemente distribuída entre 1,2 e 2,4 e atinge uma porosidade de 0,283.

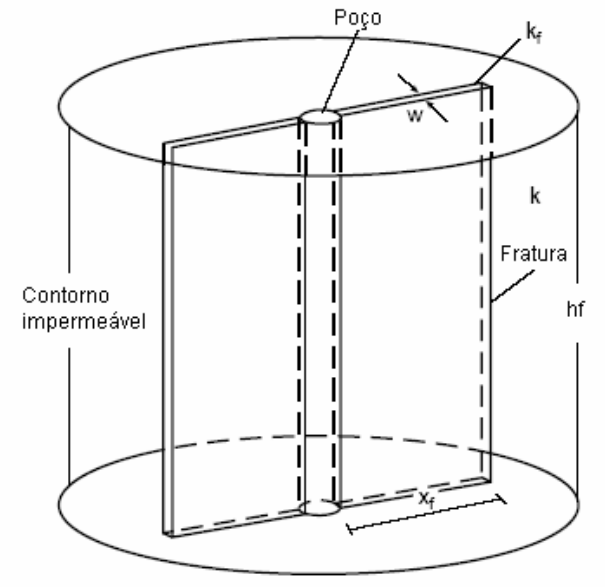

Figura 3.26: Geometria do modelo de fratura adotada (*apud* Economides e Nolte [17]).

Tabela 3.16: Resultado da geração de partículas em poços da Petrobras.

Poço	Tempo(ms)	Partículas geradas	Porosidade
7–CP–0370–SE	921	12.801	0,244
	906	12.795	0,244
	906	12.797	0,244
7–CP–1414–SE	1906	31.823	0,320
	1875	31.677	0,323
	1875	31.759	0,321
7–CP–0037–SE	2421	40.800	0,298
	2437	40.797	0,298
	2453	40.805	0,298

As figuras 3.31 e 3.32 apresentam a geração de arranjos em um domínio circular de raio 12,2mm. A primeira utiliza granulometria constante igual a 1,2mm e atinge uma porosidade de 0,277, enquanto a segunda utiliza granulometria uniformemente distribuída entre 1,2 e 2,4 e atinge uma porosidade de 0,300.

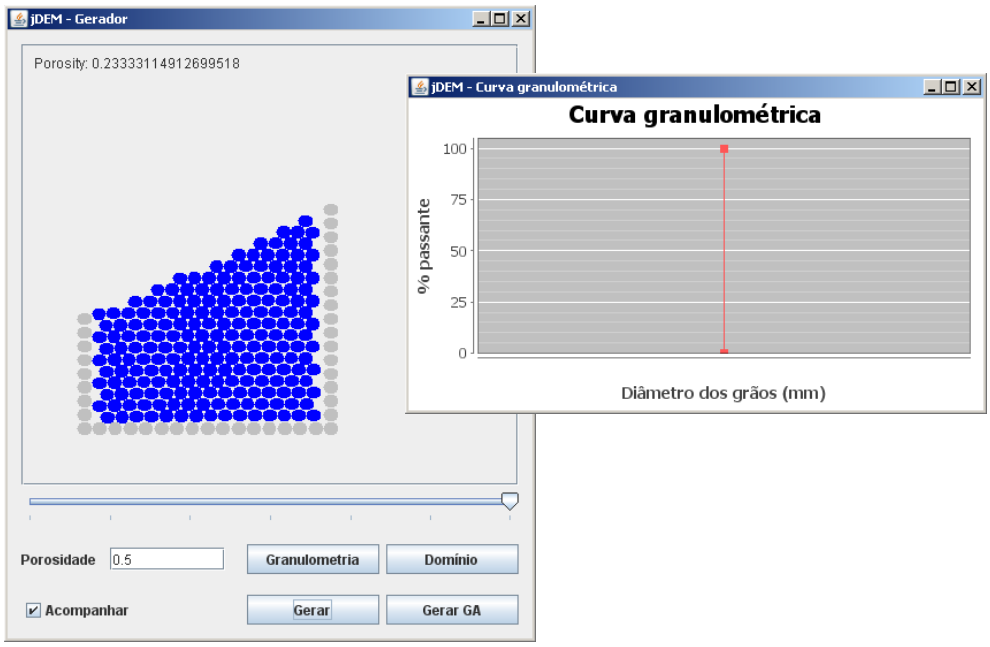

Figura 3.27: Geração de partículas em domínio trapezoidal com granulometria constante.

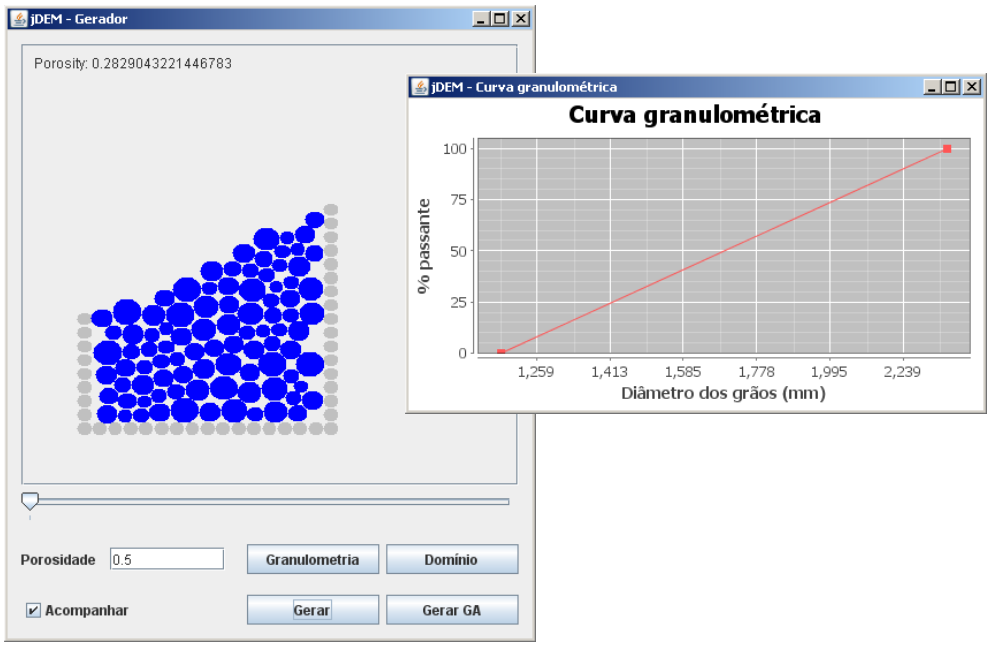

Figura 3.28: Geração de partículas em domínio trapezoidal com granulometria uniformemente distribuída.

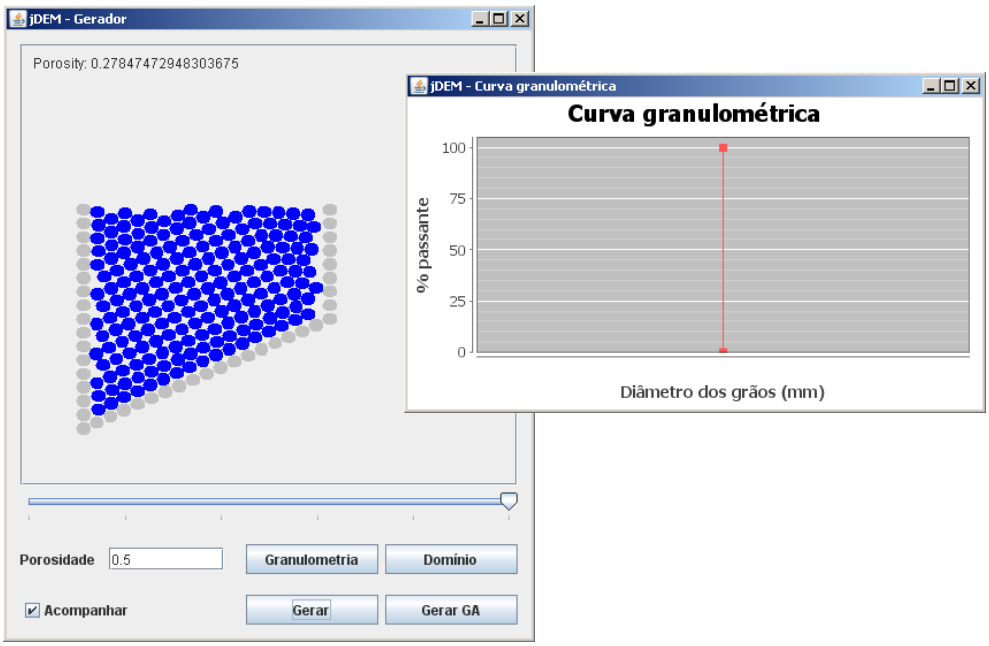

Figura 3.29: Geração de partículas em domínio trapezoidal invertido com granulometria constante.

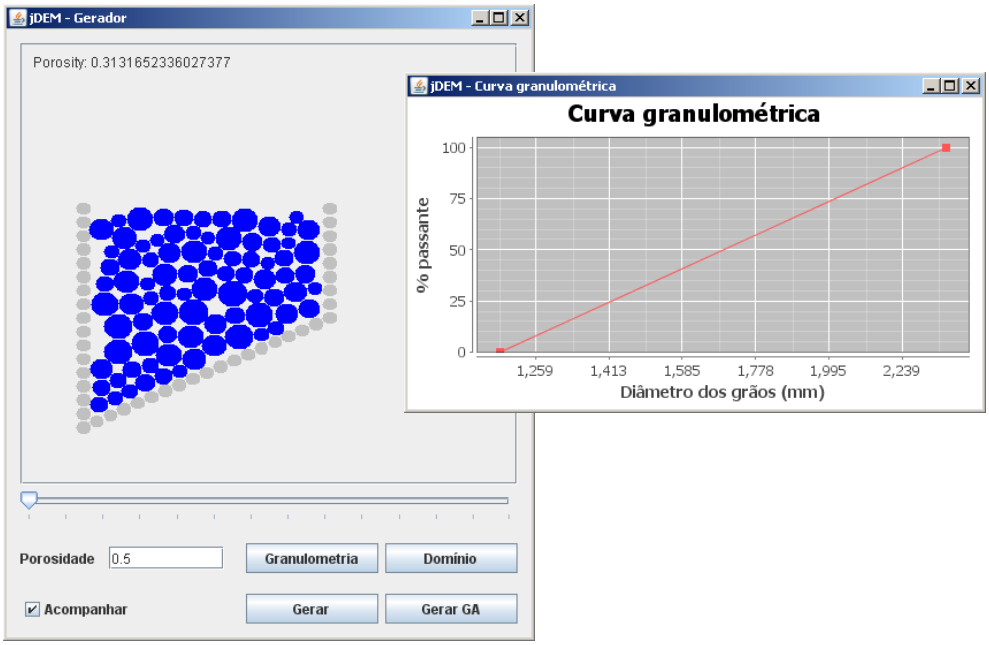

Figura 3.30: Geração de partículas em domínio trapezoidal invertido com granulometria uniformemente distribuída.

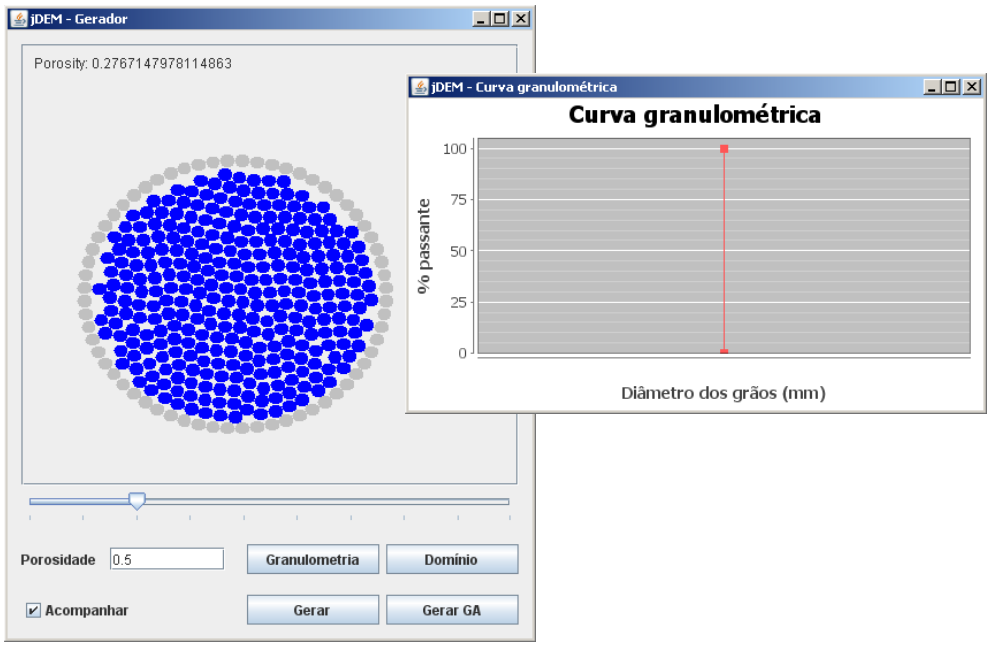

Figura 3.31: Geração de partículas em domínio circular com granulometria constante.

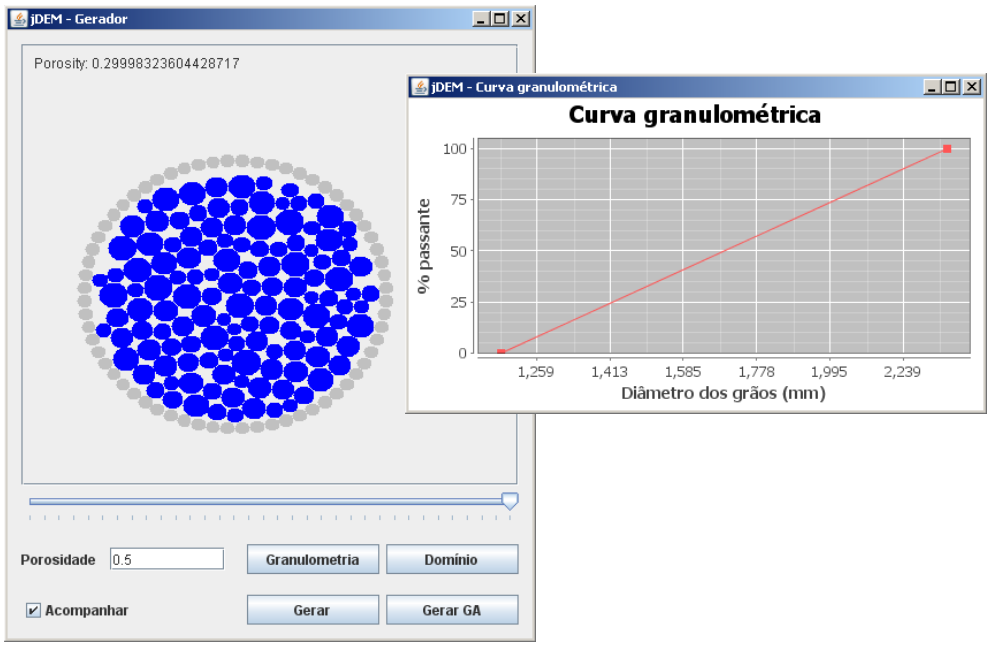

Figura 3.32: Geração de partículas em domínio circular com granulometria uniformemente distribuída.