

**Autor:** Jefter Santiago      **Projeto 4 - Modelos de Crescimento**  
**#USP:** 12559016  
**Curso:** Física Estatística Computacional      **2024.1**  
**Prof.** F. C. Alcaraz      **Data de entrega:** 04/05/20.3

---

## Autômatos celulares determinísticos (ACD)

Começamos o estudo de modelo de crescimento partindo de um modelo determinístico. Os ACDs consistem em uma regra que transforma uma configuração de uma grade (no nosso caso unidimensional) em outra, ou seja, se tivermos a configuração  $\mathcal{C}_t$  uma regra  $\mathcal{F}$  nos fornece  $\mathcal{C}_{t+1}$ .

Para uma cadeia de  $L$  sítios consideramos dois possíveis estados, 1 ou 0, e o estado futuro de um sítio dependerá dos dois sítios adjacentes<sup>1</sup>. Dessa forma, para qualquer sítio  $i$ , temos a tupla de 3 componentes que definem seu estado em  $t + 1$ . Logo, um conjunto  $\{b_{i-1}, b_i, b_{i+1}\}$  tem  $2^3$  possíveis estados, como os estados são binários, no total teremos  $2^8 = 256$  estados possíveis para um sítio.

Sabendo disso podemos gerar realizar simulações de autômatos celulares partindo de regras quaisquer e utilizando a representação binário do número para calcular o próximo estado de um sítio.

Nosso interesse é simular o automatos com 3 configurações iniciais: todos os sítios 0 , todos 1 e uma configuração aleatória. Foram feitas as simulações para as regras 232<sup>2</sup>, 254<sup>3</sup> e a regra 51.<sup>4</sup>

Segue o código implementado para os ACDs:

```

1 !     Gera estado inicial da cadeia.
2 !
3 !     Se config = 0 => C_01 = {0, ..., 0}
4 !     Se config = 1 => C_02 = {1, ..., 1}
5 !     Se config qualquer outro valor => C_03 = {b_1,..., b_L}
6 !     com b_i aleatorio.
7 subroutine set_initial_state(C, config, L)
8     implicit integer (c-c)
9     dimension C(500)
10
11    if(config == 0) then
12        C(:) = 0
13    else if (config == 1) then
14        C(:) = 1
15    else
16        p = rand(iseed)
17        do i = 1, L
18            p = rand() * 2
19            k in [0, 1]
20            k = int((p+1) / 2)
21            print *, k
22            C(i) = k
23        end do
24    end if
25    end subroutine
26
!
```

*Popula o vetor rules de 8 posições*

<sup>1</sup> Assumindo condições periódicas de contorno.

<sup>2</sup> “Regra da maioria”.

<sup>3</sup> “Regra da epidemia”.

<sup>4</sup> “Regra do contra”.

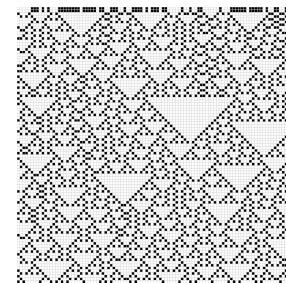


Figura 1: Regra 18.

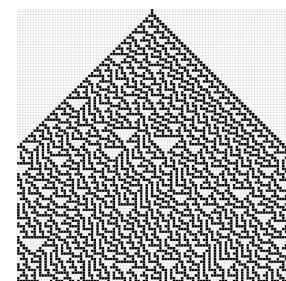


Figura 2: Regra 86

```

27 !      com valores 0 ou 1
28 !
29 !      a partir de um número inteiro
30 !      entre 0, 255
31 subroutine rule_set(rules, N)
32     implicit integer(r-r)
33     dimension rules(8)
34     M = N
35 !     print *, "N = ", N
36 !     print *, "M = ", M
37     do i = 1, 8
38         x = real(M)
39         rules(i) = mod(M, 2)
40         M = int(x/2.0)
41     end do
42
43 !     do i = 1, 8
44 !         print *, rules(i)
45 !     end do
46
47
48
49 subroutine propagate(C, rules, L)
50     implicit integer(c-c, r-r)
51     dimension rules(8)
52     ! C_{t}
53     dimension C(500)
54     ! C_{t+1}
55     dimension C_tmp(500)
56
57     do i = 2, L-1
58         C_tmp(i) = rules(4*C(i-1)+2*C(i)+C(i+1)+1)
59     end do
60
61     C_tmp(1) = rules(4*C(L)+2*C(1)+C(2)+1)
62     C_tmp(L) = rules(4*C(L-1)+2*C(L)+C(1)+1)
63
64     C = C_tmp
65
66 end subroutine
67
68
69 ! Max size of chain; L = 500
70 subroutine DCA(rule_number, N_iter, L, init_state, f_name)
71     implicit integer(f-f, c-c, r-r)
72     dimension rules(8)
73     dimension C(500)
74
75     print *, "rule_number = ", rule_number
76     call set_initial_state(C, init_state, L)
77
78     write(f_name, *) (C(j), j = 1, L)
79
80     call rule_set(rules, rule_number)
81
82     do i = 1, N_iter

```

```

83     write(f_name, *) (C(j), j = 1, L)
84     call propagate(C, rules, L)
85 end do
86 end subroutine DCA
87
88 ! Max size of chain; L = 500
89 subroutine DCA_position(rule_number, N_iter, L, pos, f_name)
90 implicit integer(f-f, c-c, r-r, p-p)
91 dimension rules(8)
92 dimension C(500)
93
94 !      print *, "rule_number = ", rule_number
95 call set_initial_state(C, 0, L)
96 C(pos) = 1
97
98 write(f_name, *) (C(j), j = 1, L)
99
100 call rule_set(rules, rule_number)
101
102 do i = 1, N_iter
103     write(f_name, *) (C(j), j = 1, L)
104     call propagate(C, rules, L)
105 end do
106 end subroutine DCA_position

```

Essas rotinas estão no arquivo tarefa-1/cellular-automaton.f e são utilizadas no programa principal tarefa-1/tarefa-1.f:

```

1 implicit integer (c-c, r-r)
2 ! C_{t}
3 L = 100
4 K = 100
5 print *, "Forneca regra R [0, 255]"
6
7 read(*, *) R
8
9 open(unit = 1, file="rule-#-0.dat")
10 open(unit = 2, file="rule-#-1.dat")
11 open(unit = 3, file="rule-#-2.dat")
12
13 call DCA(R, K, L, 0, 1)
14 call DCA(R, K, L, 1, 2)
15 call DCA(R, K, L, 2, 3)
16
17 close(1)
18 close(2)
19 close(3)
20 end

```

Segue abaixo as imagens da evolução dos automatos celulares executados pelo tarefa-1/tarefa-1.f:

Além desses padrões, também feito uma variação do programa principal e um script externo em *bash* que executa o programa para todas as regras possíveis de 0 à 255. Esse script é o tarefa-1/all-rules.sh e deve ser tornado executável para executar o programa tarefa-1/all-rules.f. As configurações são salvas no diretório saidas/tarefa-1/all-rules/

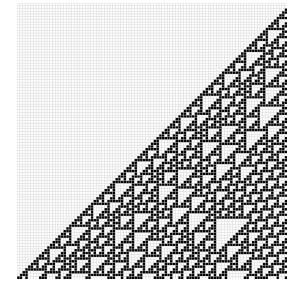


Figura 3: Regra 110.

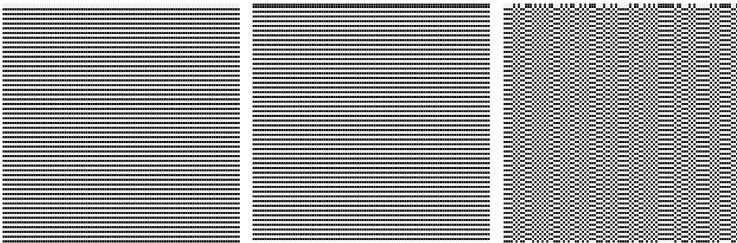


Figura 4: Regra 51.

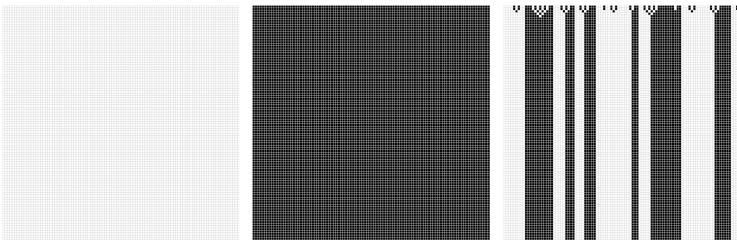


Figura 5: Regra 232.

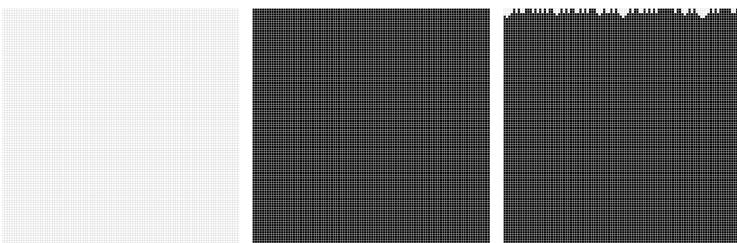


Figura 6: Regra 254.

e além disso os gráficos ficam em uma pasta de mesmo nome no diretório `graficos/`. Algumas das outras regras estão nas figuras ??, ??, ??.

## DLA 2D

O principal modelo de crescimento estudado foi do *Diffusion limited aggregation*, que é um modelo de crescimento aleatório. Nesse modelo deixamos utilizamos de random walks para observar um padrão de crescimento para um agloramerado de partículas. Além implementar a simulação e observar graficamente o resultado final do aglomerado, também foi realizado o cálculo da dimensão fractal associada à esse tipo de crescimento.

Na simulação centramos nossa semente inicial sempre na origem do sistema de coordenadas e então adotamos um raio  $R_i$  no qual outras partículas devem inicializar seu movimento aleatório. Partículas que excedem uma distância maior que um fator  $\beta = 1.5$  do tamanho do raio inicial são descartadas<sup>5</sup>. Partículas que atingem o agregado passam a compô-lo. Além disso, sempre que adicionamos novas partículas ao corpo é feita uma checagem sobre o tamanho do raio atual, se necessário este é atualizado.

À cada adição de novas partículas ao agregado, também somamos o número de partículas total para o raio atual, dessa forma no arquivo de saída do programa temos dados para observar o crescimento do número de partículas em função do raio  $N(r)$  e determinar qual a dimensão fractal da simulação.

Segue abaixo o código para implementação do DLA-2D.<sup>6</sup>

```

1 !      Dinâmica DLA em 2D
2 !
3 !      Para cada partícula:
4 !
5 !          - Gera posição inicial aleatoria.
6 !
7 !          - Aplica dinâmica de random walk.
8 !
9 !          - Ao agregar novas celulas armazena a posição e raio atual à um
10 !         arquivo de dados.
11 !
12 !         Parâmetros:
13 !
14 !         Np -> numero de partículas
15 subroutine DLA_2D(Np, iseed)
16     implicit integer (x-x, y-y)
17     parameter(N = 500)
18     dimension lattice(-N:N, -N:N)

19 !
20 !         semente inicial
21     lattice(0, 0) = 1

22 !
23     rnd = rand(iseed)

24 !
25     R_in = 5.0
26     R_f = 1.5 * R_in

27 !
28     open(1, file="saída-dla.dat")
29     open(2, file="saída-contagem.dat")

30 !
31     nparts = 0

32 !
33     do i = 1, Np

34         x = 0
35         y = 0

```

<sup>5</sup> É a simulação delas iniciada novamente.

<sup>6</sup> Assim como anteriormente, o código é um conjunto das subrotinas e o programa principal é tarefa-2/tarefa-2.f|.

```

31
32     print *, "Particle #", i
33
34     call generate_random_particle(R_in, x, y)
35
36     s = 0
37     touched = 0
38
39     do while(touched == 0)
40
41         call random_step(x, y)
42
43         d = sqrt(real(x**2 + y ** 2))
44
45 !     Captura celula ao redor da posição atual se houver
46         do k = -1, 1
47             do j = -1, 1
48                 ! Checagem de borda
49                 if(abs(x) < N .and. abs(y) < N)then
50                     s = s + lattice(x+k, y+j)
51                 end if
52             end do
53         end do
54
55 !     Adiciona nova celula, atualiza raio
56         if (d >= R_f) then
57             touched = 1
58         else if(s >= 1) then
59             touched = 1
60
61 !     Adiciona mais uma particula à conta do raio.
62         lattice(x, y) = 1
63
64 !     Salva o cluster
65         nparts = nparts + 1
66         write(1, *) x, y, R_in
67         write(2, *) R_in, nparts
68
69         if(d > R_in) then
70             R_in = d + 5
71             R_f = 1.5 * R_in
72         end if
73     end if
74
75 end do
76
77 close(2)
78
79 end subroutine DLA_2D
80
81 !     Gera uma partícula em uma posição aleatoria
82 !     dado um raio inicial R_in
83 subroutine generate_random_particle(R_in, x, y)
84     implicit integer (c-c, x-x, y-y)
85     parameter(pi = acos(-1e0))
86
87     rnd_val = rand()
88
89     theta = 2 * pi * rnd_val
90
91     x = int(R_in * cos(theta))
92     y = int(R_in * sin(theta))

```

```

88      end subroutine generate_random_particle
89
90 !     Dinâmica das partículas.
91 !     Executa random-walk até que: atinge uma célula ocupada
92 !     Parâmetros:
93 !     posição x, y
94     subroutine random_step(x,y)
95         implicit integer(x-x,y-y)
96
97         x = x + floor(rand()*3) - 1
98         y = y + floor(rand()*3) - 1
99
100        end subroutine random_step

```

O programa principal é simplesmente:

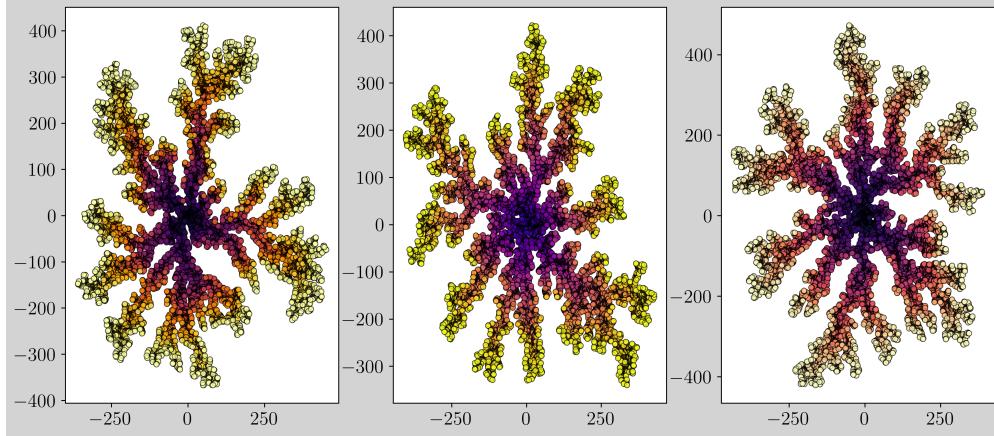
```

1 ! DLA (2D)
2 read(*, *) iseed
3 call DLA_2D(50000, iseed)
4 end

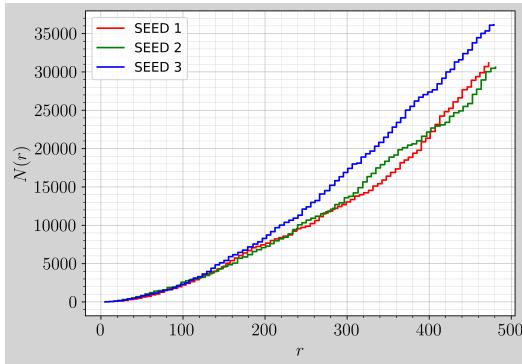
```

Como na tarefa anterior, também foi criado um arquivo .sh para executar o tarefa-2.exe para alguns diferentes *seeds* e mover os outputs para o diretório de saídas.

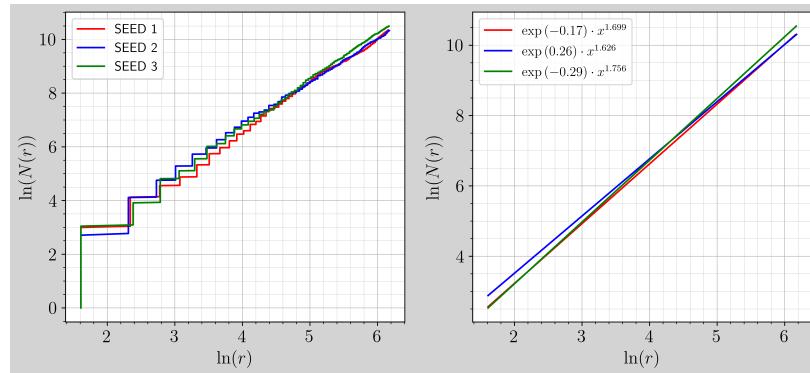
A figura (??) é o resultado da simulação para algumas seeds de número aleatório diferentes. O gráfico de calor foi feito a partir da magnitude da distância normalizada para cada um dos plots, sendo os pontos mais escuros mais próximos da semente inicial da simulação e mais claros são mais distantes.



Analisando puramente a contagem do número de pontos em um dado raio podemos obter o seguinte gráfico(??):



Colocando em  $\log \times \log$  e realizando regressão linear obtemos a dimensão fractal  $d_f$  para cada uma das simulações.



Isto, é, para a SEED 1 temos  $d_f \approx 1,69$ , SEED 2  $d_f \approx 1,64$  e SEED 3  $d_f \approx 1,76$ . Esses valores estão dentro do esperado para o DLA em duas dimensões.

Figura 7: DLA 2D: Configuração final dos agregados para semenetes 169, 255 e 954, respectivamente.

Figura 8: Número de pontos em relação ao raio.

Figura 9: Gráfico da dimensão fractal em  $\log \times \log$  e regressão linear.

## DLA 3D

O modelo *DLA* em 3D não oferece nenhuma grande dificuldade de implementação, já que podemos aproveitar boa parte do código implementado no anterior. É esperado que a dimensão fractal seja maior que a do modelo anterior com  $2 < d_f < 3$ .

As mudanças do código *DLA-2D.f* para o *DLA-3D.f* consistem em adição de mais uma coordenada e assim como antes também há um arquivo *run.sh* para executar algumas *seeds* pre-determinadas (no próprio arquivo).

Segue abaixo o código que implementa a dinâmica do *DLA-3D*:

```

1 !      Dinâmica DLA em 2D
2 !
3 !      Para cada partícula:
4 !
5 !      - Gera posição inicial aleatória.
6 !
7 !      - Aplica dinâmica de random walk.
8 !
9 !      - Ao agregar novas células armazena a posição e raio atual à um
10 !
11 !     arquivo de dados.
12 !
13 !     Parâmetros:
14 !
15 !     Np -> numero de partículas
16 subroutine DLA_3D(Np, iseed)
17 implicit integer (x-x, y-y, z-z)
18 parameter(N = 500)
19 dimension lattice(-N:N, -N:N, -N:N)
20
21
22
23 !     semente inicial
24 lattice(0, 0, 0) = 1
25
26
27 rnd = rand(iseed)
28
29 R_in = 5.0
30 R_f = 1.5 * R_in
31
32
33 open(1, file="saída-dla.dat")
34 open(2, file="saída-contagem.dat")
35
36 nparts = 0
37 do i = 1, Np
38
39     x = 0
40     y = 0
41     z = 0
42
43     print *, "Particle #", i
44
45     call generate_random_particle(R_in, x, y, z)
46
47     print *, "r = ", R_in
48     s = 0
49     touched = 0
50
51     do while(touched == 0)
52
53         call random_step(x, y, z)
54
55         if (lattice(x,y,z) == 0) then
56             lattice(x,y,z) = 1
57             s = s + 1
58             if (s .gt. R_f) then
59                 touched = 1
60             end if
61         end if
62
63         call random_step(x, y, z)
64
65     end do
66
67     print *, "Número de partículas: ", nparts
68     print *, "Raio final: ", R_f
69
70     print *, "Número de partículas: ", nparts
71     print *, "Raio final: ", R_f
72
73     write(1, *) nparts, R_f
74
75     nparts = nparts + 1
76
77 end do
78
79
80 close(1)
81 close(2)
82
83
84
85
86
87
88
89
90
91
92
93
94

```

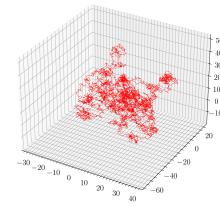


Figura 10: Ilustração de uma caminhada aleatória em 3D.

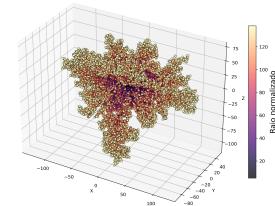


Figura 11: SEED = 51

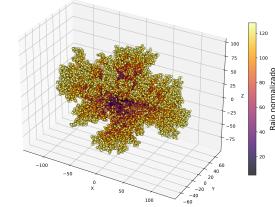


Figura 12: SEED = 255

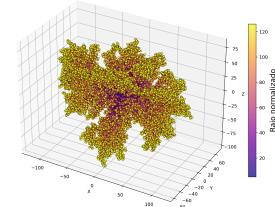


Figura 13: SEED = 729

```

45         d = sqrt(real(x**2+y**2+z**2))
46 !   Captura celula ao redor da posição atual se houver
47         do k = -1, 1
48             do j = -1, 1
49                 do l = -1, 1
50 !     Checagem de borda
51         if(abs(x) < N .and. abs(y) < N)then
52             s = s + lattice(x+k, y+j, z+l)
53         end if
54     end do
55 end do
56 end do
57 !   Adiciona nova celula, atualiza raio
58     if (d >= R_f) then
59         touched = 1
60     else if(s >= 1) then
61         touched = 1
62         lattice(x, y, z) = 1
63
64     nparts = nparts + 1
65 !   Salva o cluster, particulas e raio
66     write(1, *) x, y, z, R_in
67     write(2, *) R_in, nparts
68     if(d > R_in) then
69         R_in = d + 5
70         R_f = 1.5 * R_in
71     end if
72     end if
73 end do
74 end do
75 close(1)
76 close(2)
77 end subroutine DLA_3D
78 !   Gera uma partícula em uma posição aleatória
79 !   dado um raio inicial R_in
80 subroutine generate_random_particle(R_in, x, y, z)
81 implicit integer (x-x, y-y, z-z)
82 parameter(pi = acos(-1e0))
83
84 rnd_val1 = rand()
85 rnd_val2 = rand()
86
87 theta = 2 * pi * rnd_val1
88 phi = 2 * pi * rnd_val2
89
90 x = int(R_in * cos(theta)*cos(phi))
91 y = int(R_in * sin(theta)*sin(phi))
92 z = int(R_in * sin(theta))
93
94 end subroutine generate_random_particle
95
96 !   Dinâmica das partículas.
97 !   Executa random-walk até que: atinge uma celula ocupada
98 !   Parâmetros:
99 !   posição x, y
100 subroutine random_step(x, y, z)

```

```

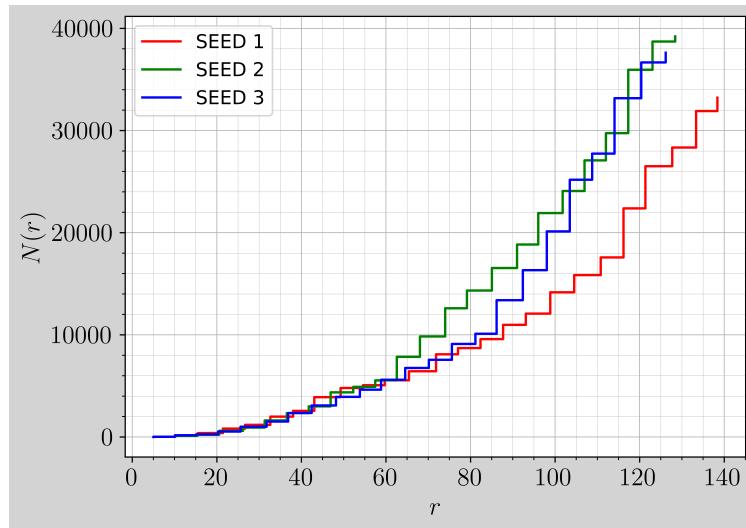
101 implicit integer(x-x,y-y,z-z)

102
103 x = x + floor(rand()*3) - 1
104 y = y + floor(rand()*3) - 1
105 z = z + floor(rand()*3) - 1
106
107 end subroutine random_step

```

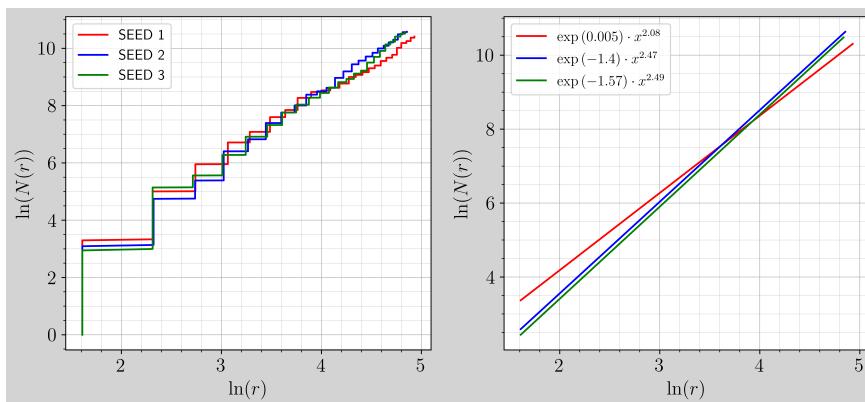
As imagens(??)(??)(??) dos aglomerados em 3D foram adicionados apenas como fins de ilustração. No entanto usando alguns software interativos pode ser útil ter uma representação 3D do corpo final.

Assim antes estamos interessados em observar o crescimento de numero de pontos para um dado raio. O resultado obtido está na figura abaixo (??).



Fazendo novamente a linearização e regressão linear obtemos os coeficientes que determinam a dimensão fractal para as simulações.

Figura 14: Número de pontos  $N(r)$  em função do raio  $r$ .



Pela (??) obtemos  $d_f = 2,08$ ,  $d_f = 2,47$  e  $d_f = 2,49$  para as seeds de número aleatório utilizadas. Não foram feitas análises mais detalhadas dos valores obtidos como médias ou desvio padrão.

Figura 15: Estimando dimensão fractal  $d_f$ .

## Efeito corona

Nessa tarefa aplicamos o modelo de crescimento implementado na tarefa dois para *DLA-2D* em condições iniciais nas quais o comportamento do sistema é parecido o efeito corona observado em fios condutores.

As únicas alterações realizadas ao código `tarefa-2/DLA_2D.f` foram as condições iniciais e geração de pontos aleatórios. Adotamos sistema com partículas situadas em todas posição  $x$  para dado vértice  $y$ , que para simulação foi adotado sendo  $y = 0$ . Os pontos aleatórios que realizam movimento browniano são gerados como antes à uma distância fixa, mas não são pontos aleatórios no plano ou espaço, ao invés de angulo adotamos um espaço retangular no qual o ponto pode ser gerado.

Segue abaixo o código da segunda tarefa alterado para simular o efeito corona:

```

1 !      Efeito corona
2 !
3 !      código da tarefa-2 : dla_2d
4 !      com adaptação nas condições iniciais e tamanho
5 !      do lattice em Y.
6
7     implicit integer (x-x, y-y)
8     parameter(N = 800)
9     dimension lattice(-N:N, 0:5000)
10    Np = 80000
11    read(*, *) iseed
12    call strand(iseed)
13    lattice = 0
14    open(1, file="saida-dla.dat")
15    !      semente inicial
16    do i = -N, N
17      lattice(i, 0) = 1
18      write(1, *) i, 0
19    end do
20    Y_in = 5
21    Y_f = 1.5 * Y_in
22    do i = 1, Np
23      touched = 1
24      print *, "Particle #", i
25      !      Gera uma partícula em um ponto aleatório.
26      x = (2 * N * rand()) - N
27      y = R_in
28      do while(touched == 1)
29        !      Passo aleatório
30        x = x + floor(rand()*3) - 1
31        y = y + floor(rand()*3) - 1
32      !      Conta número de vizinhos próximos
33      do j = -1, 1
34        do k = 0, 1
35          sum = sum + lattice(x + j, y - k)
36        end do
37      end do
38      if(x >= N .or. x < -N .or. y > R_f) then
39        touched = 0
40      else if (sum > 0) then

```

```

39      lattice(x, y) = 1
40      write(1, *) x, y
41      touched = 0
42      sum = 0
43      if(y == R_in) then
44          R_in = R_in + 5
45          R_f = R_in * 1.5
46      end if
47      end if
48  end do
49 end do
50 close(1)
51 end

```

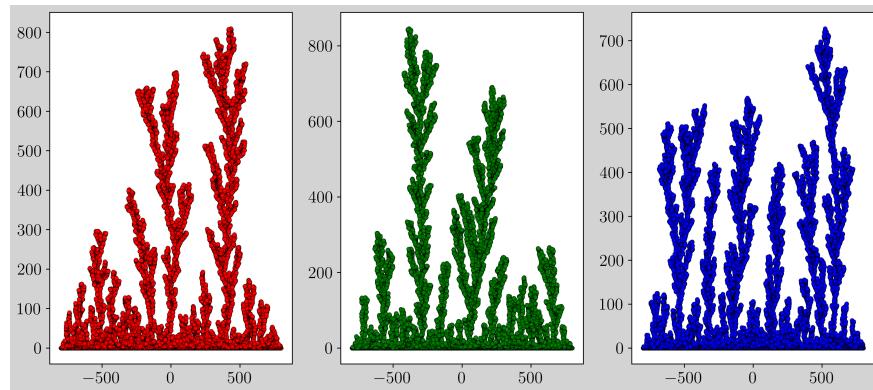


Figura 16: Simulações do Efeito Corona para algumas *seeds* quaisquer.

## Revoluções populares

A última, e mais desafiadora, simulação realizada é a de um modelo simples de “revolução popular” utilizando as ideias do *diffusion limited aggregation*.

Nesse modelo geramos por um processo de Bernoulli um número considerável de partículas que permanecem estáticas em seus respectivos sítios do reticulado.<sup>7</sup> Posicionamos então uma partícula que realizará random walk com a seguinte regra: ao atingir um reticulado ocupado, ou seja, que tenha estado diferente de zero, deve adicionar essa partícula ao aglomerado. Na simulação podemos limitar a contagem de passos que o “processo revolucionário” vai realizar e observar o estado final do sistema após os  $N$  passos. É do nosso interesse observar a dependência do tamanho da dimensão fractal  $d_f$  com a probabilidade  $p$  adotada no processo de Bernoulli.

Foi adotada uma estratégia força bruta para realizar a simulação. Utilizando dois *grids* principais que representam o reticulado estático e o dinâmico, podemos partir da origem do sistema com uma partícula e percorrer um raio  $R$  em caminhada aleatória. Isto é, buscamos em torno de um raio inicial se há partículas adjacentes, caso haja, são adicionadas e realizamos checagem do tamanho atual do raio e a distância da partícula recém adicionada à o centro. Isso é semelhante ao que foi feito na simulação do *DLA-2D*. Parte crucial da dinâmica é a realização do movimento aleatório de todo o agregado. Para esse fim, temos um loop duplo em torno do raio  $R$  atual no grid dinâmico que analisa qual sítio do reticulado possui uma partícula, caso haja, essa partícula realiza um passo aleatório. Todas outras partículas do agregado realizam o mesmo passo.

Uma estratégia mais eficiente talvez seria buscar uma forma de realizar o movimento sem necessitar analisar se há partículas em cada um dos sítios, utilizando alguma estrutura que salvasse estado atual dos entornos.<sup>8</sup>

Segue abaixo o código implementado para a simulação do modelo de revolução popular:

```

1      implicit integer (x-x, y-y)
2      parameter(N = 1000)

3
4      dimension lattice_static(-N:N, -N:N)
5      dimension lattice_dynamic(-N:N, -N:N)
6      dimension lattice_aux(-N:N, -N:N)

7
8      dimension ipx(0:3)
9      dimension ipy(0:3)
10     parameter(ipx=(/1,-1,0,0/), ipy = (/0,0,1,-1/))

11
12    p = 0.1

13
14    open(1, file="output-fractal.dat")

```

<sup>7</sup> Nota-se que o sistema poderia ser ainda mais complexo se aceitassemos todas partículas se movendo simultaneamente no reticulado.

<sup>8</sup> Não consegui trabalhar nessa tarefa por tempo o suficiente para fazer um bom trabalho.

```

16 lattice_static(0, 0) = 0
17 lattice_dynamic(0, 0) = 1
18
19 call strand(33519)
20 do i = -N, N
21   do j = -N, N
22     if (rand() <= p) then
23       lattice_static(i,j) = 1
24     end if
25   end do
26 end do
27 !
28 !   Dinamica
29 !   · Agragado pode aglutinar partículas por até M passos
30 !   · Definimos um raio para o agregado pode buscar partículas adjacentes
31 !   · Para cada passo olhamos em volta do agragado, aglutinamos
32 !   · partículas, atualizamos raio e contamos # partículas no dado raio.
33 k_radious = 5
34 x = 0
35 y = 0
36 n_parts = 0
37 ! passos
38 do m = 1, 3500
39   print *, "STEP #", m
40   ! Percorre raio em torno das coordenadas (x,y) do aglomerado.
41   do i = -k_radious, k_radious
42     do j = -k_radious, k_radious
43       if(abs(x+i) < N .and. abs(y+j)<N) then
44         if(lattice_dynamic(x+i, y+j) == 1) then
45           ! Percorre coordenadas adjacentes ao aglomerado.
46           do k = -1, 1
47             do l = -1, 1
48               ! Aglomerado chocou com uma partícula.
49               if(abs(i+k) < N .and. abs(j+l) < N) then
50                 if(lattice_static(i+k,j+l)==1) then
51                   lattice_static(i+k,j+l) = 0
52                   lattice_dynamic(i+k,j+l) = 1
53                 ! Verifica se o raio ultrapassou do raio inicial k_radious:
54                 dist=sqrt(real((i+k-x)**2+(j+l-y)**2))
55                 if(dist >= k_radious) then
56                   k_radious = k_radious + 5
57                 end if
58               n_parts = n_parts + 1
59             ! Escreve raio atual e numero de partículas
60             write(1, *)n_parts,(i+k)-x,(j+l)-y,dist
61           end if
62         end if
63       end do
64     end if
65   end if
66 end do
67 end do
68 ! Movimenta o aglomerado:
69 x_step = 0
70 y_step = 0
71 ia = 4 * rand()

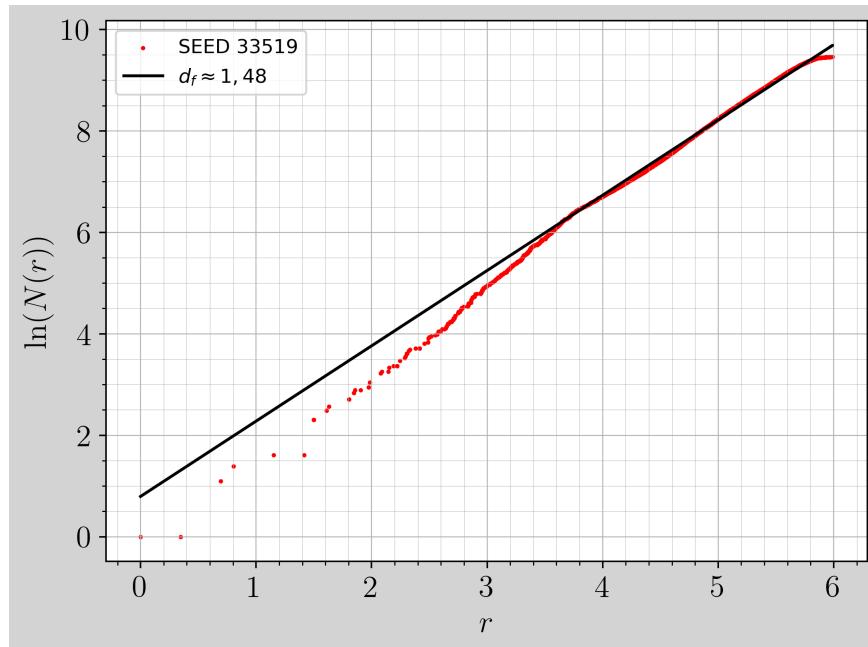
```

```

72 !      . Todas particulas do aglomerado estao dentro do raio k_radious
73   do i = x-k_radious, x+k_radious
74     do j = y-k_radious, y+k_radious
75       if(lattice_dynamic(i, j) == 1) then
76         lattice_aux(i+ipx(ia),j+ipy(ia))=lattice_dynamic(i,j)
77       end if
78     end do
79   end do
80   x = ipx(ia)
81   y = ipy(ia)
82   lattice_dynamic = lattice_aux
83   lattice_aux = 0
84 end do
85 end

```

Pela implementação feita não achei seria simples fazer diagrama com resultado final do sistema.



Como podemos ver pela (??) o valor encontrado para a dimensão  $d_f \approx 1.5$  não condiz com o esperado  $d_f \approx 1.7$ . Isso porque a forma como contei, externamente, no graficador é menos precisa que a forma que fiz anteriormente, nas tarefas 2 e 3. A minha solução para o problema ficou bastante instável e realizar não quis adicionar maior complexidade ao código para realizar a contagem como anteriormente.

Figura 17: Ajuste linear para  $p = 0.1$ .