# Self-organized criticality

*Phys 750 Lecture 4*

# Self-organized criticality

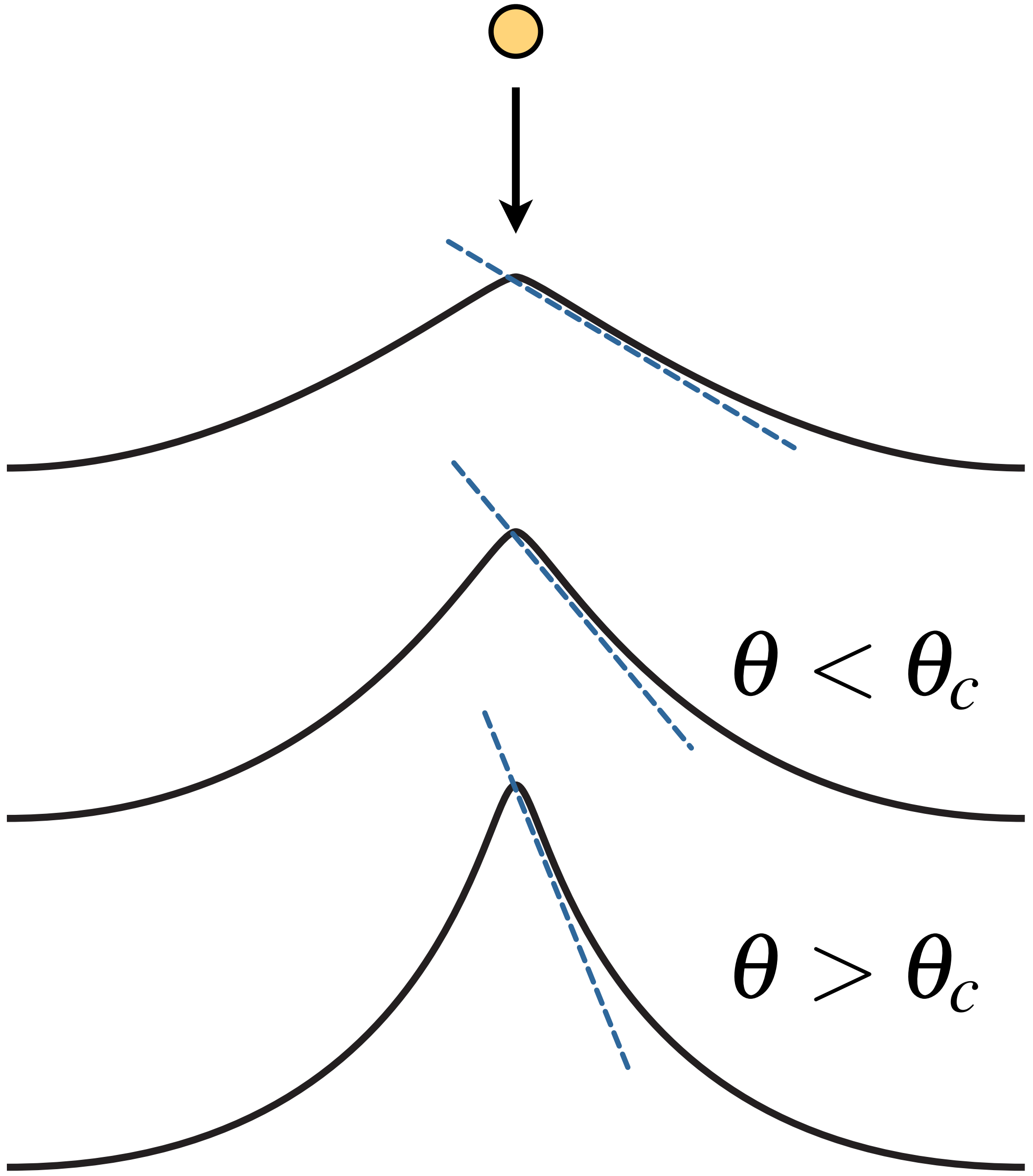
- Some characteristics:
  - system is marginally stable
  - prone to dramatic avalanche or cascade behaviour at unpredictable moments
  - power-law correlations
  - fractal size distributions

*related to scale invariance*

# Self-organized criticality

- Arises in cellular automata for earthquakes, landslides, snowflakes, epidemics, war, stock markets, ...
- But these models all seem to require some degree of randomness:
  - CA rules themselves have a probabilistic character
  - or updates are deterministic but performed asynchronously on randomly-selected cells

# Sandpile model



- Sandpile grows as we drop additional grains
- Evolves smoothly until a critical threshold is reached
- Catastrophic rearrangement plus additional cascading events

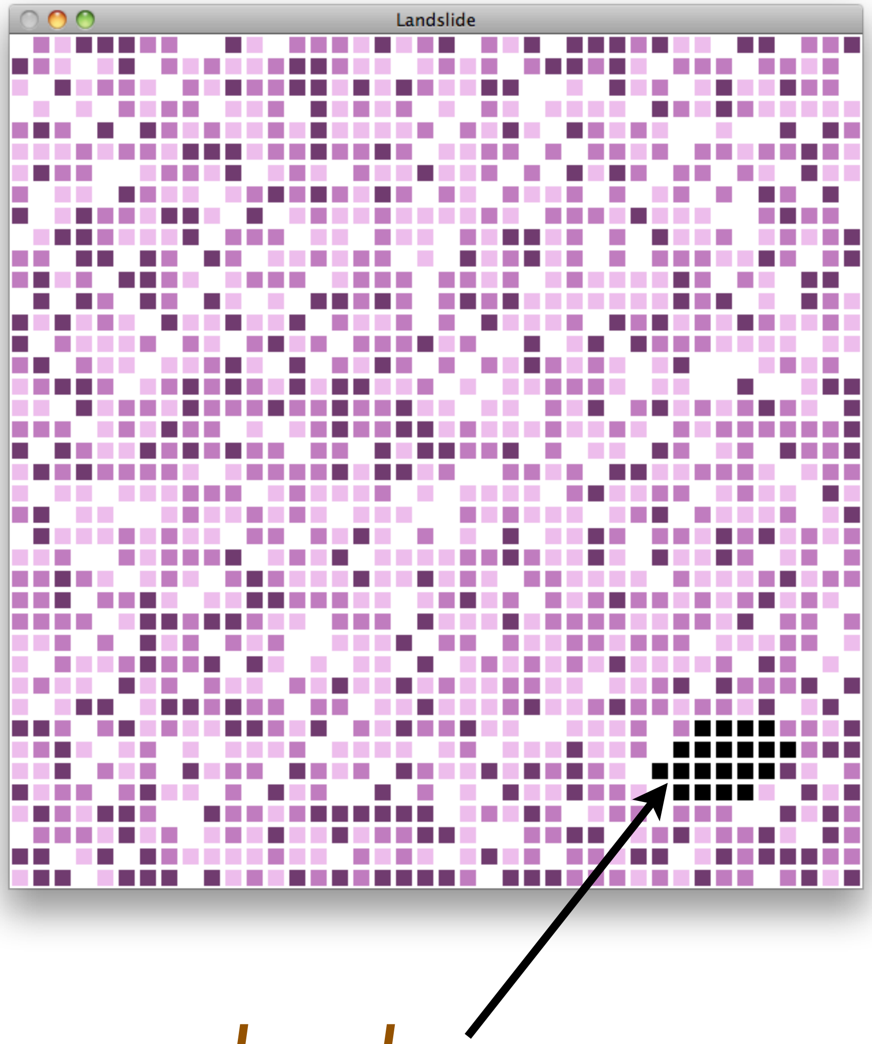
# Sandpile CA

- choose a cell $(x, y)$ at random
- increment its height by one:
  $h(x, y) := (x, y) + 1$
- if $h(x, y) = 4$ then set
  - $h(x, y) := 0$
  - $h(x+1, y) := h(x+1, y) + 1$
  - $h(x-1, y) := h(x-1, y) + 1$
  - $h(x, y+1) := h(x, y+1) + 1$
  - $h(x, y-1) := h(x, y-1) + 1$
- apply the update recursively to every height-4 neighbour

# Sandpile CA

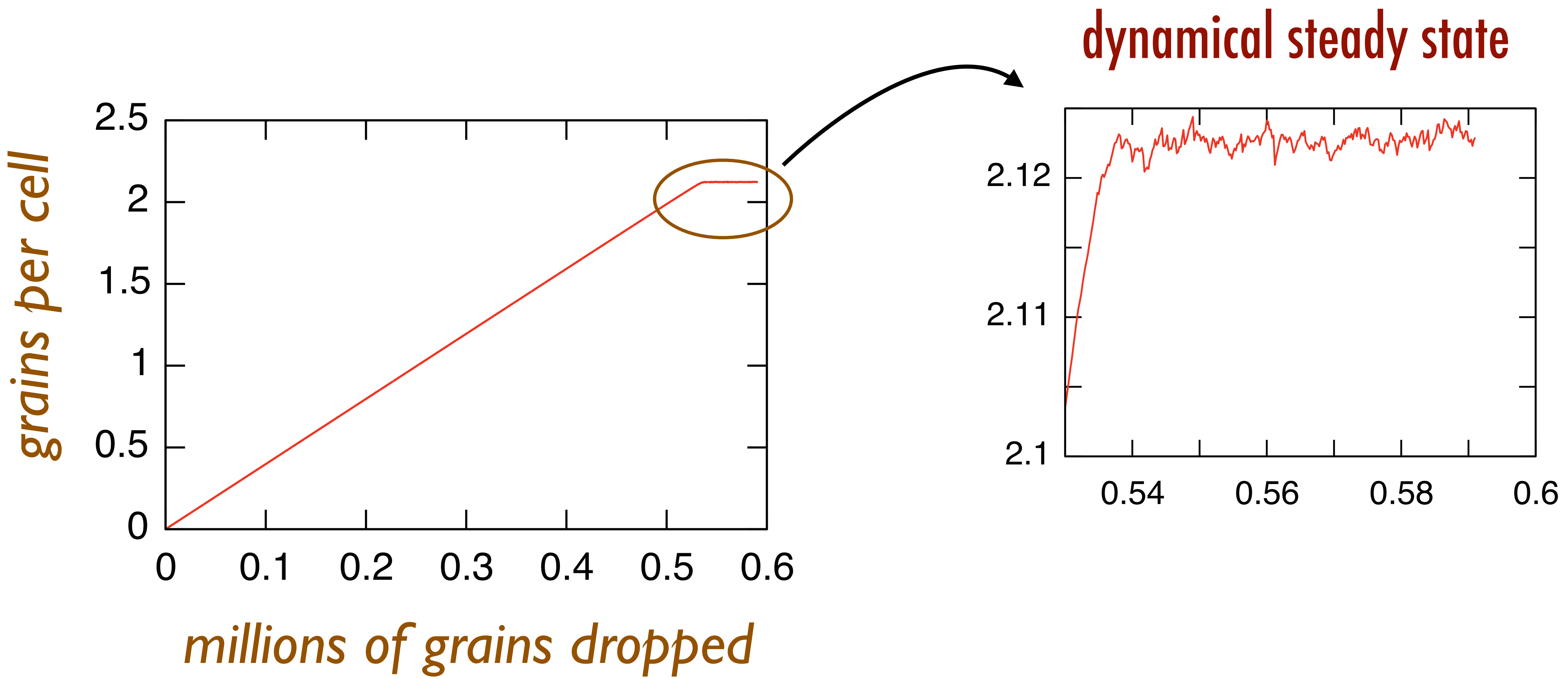
early stages of adding grains

dynamical steady state



*avalanche*

# Sandpile CA

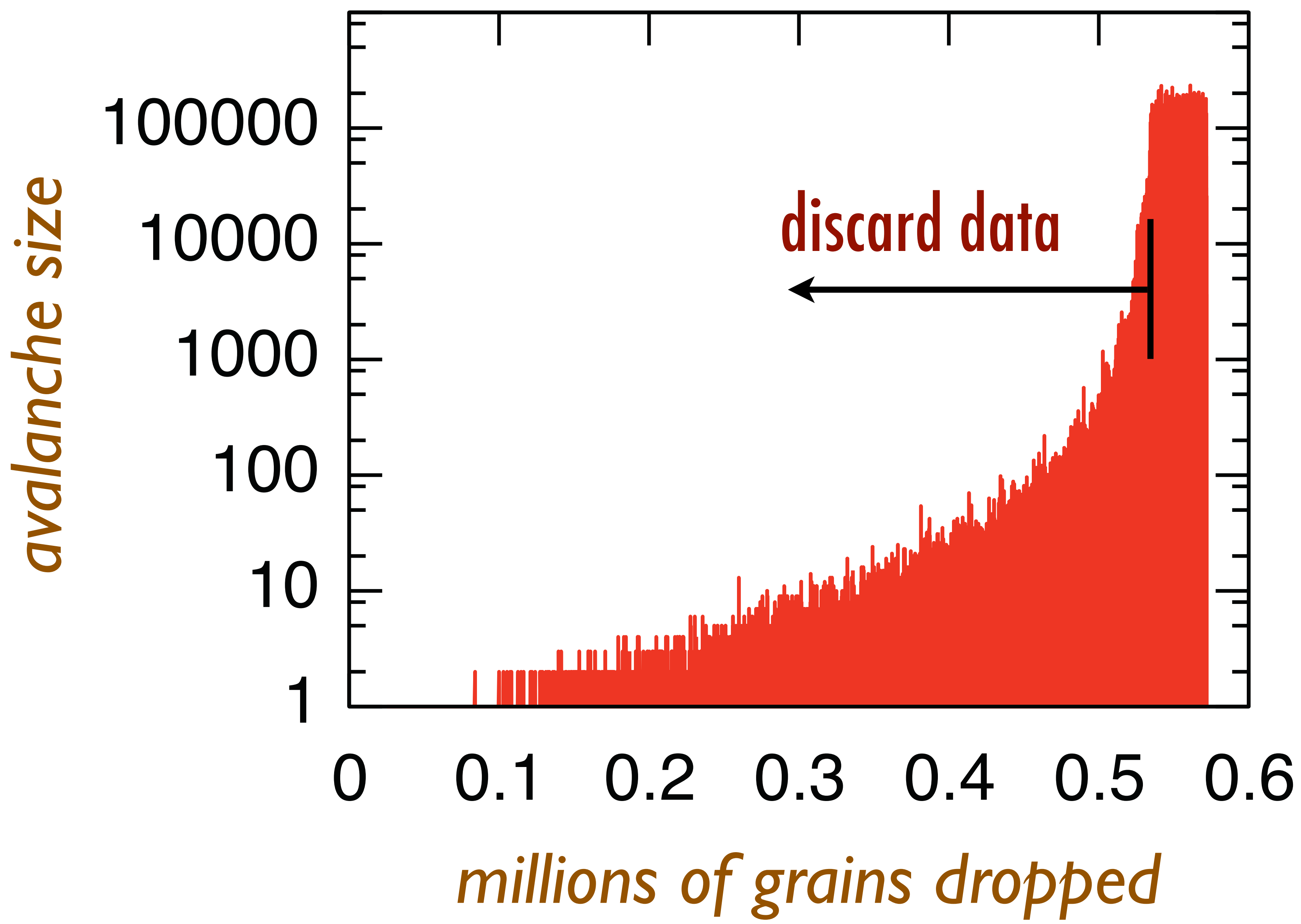


dynamical steady state

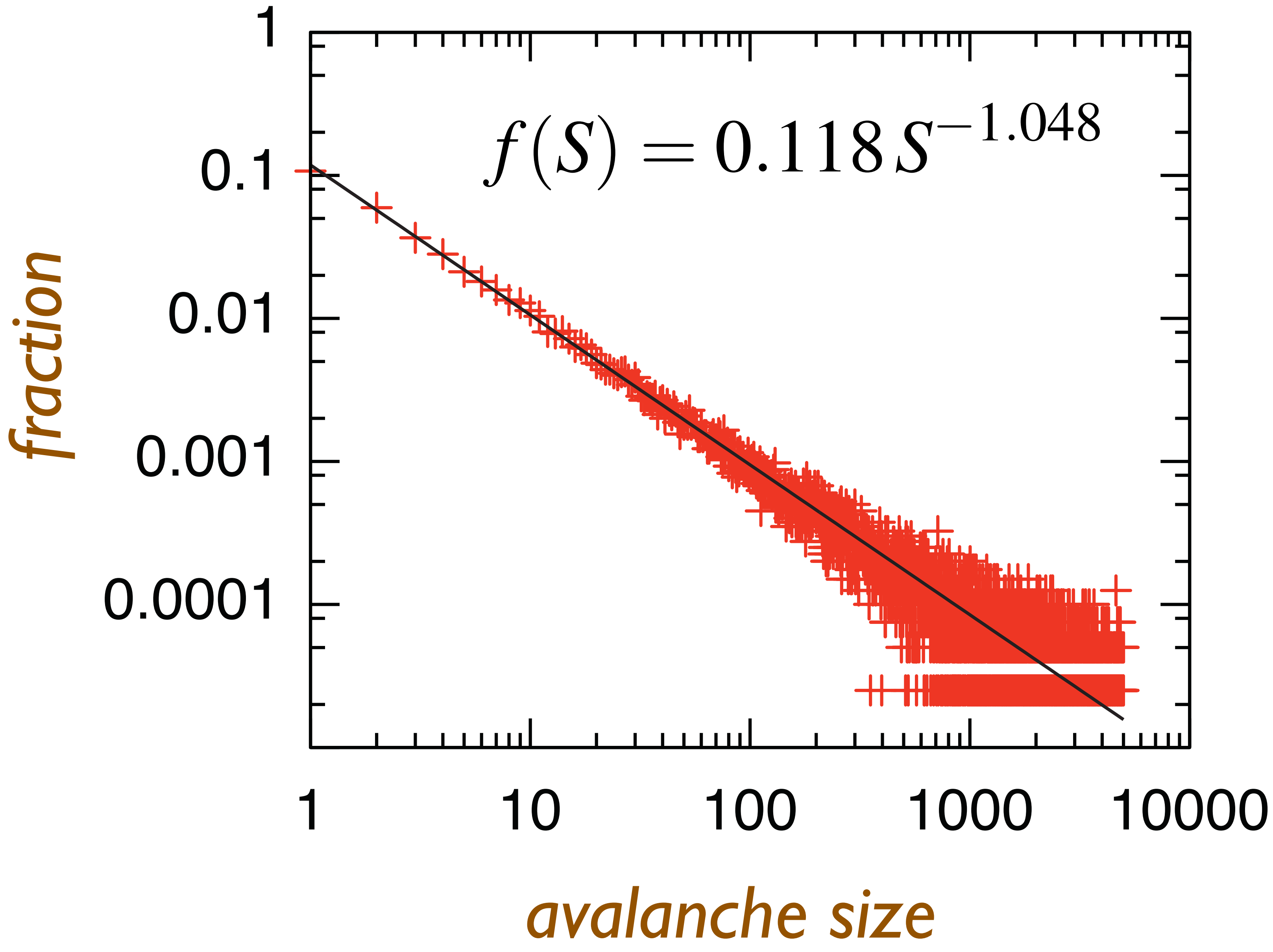
# Distribution functions

- Measure and histogram quantities in the dynamical steady state:
  - number of grains added between avalanche events
  - avalanche size
- Most quantities display power-law behaviour
- No fundamental scales in the model

# Avalanche distribution

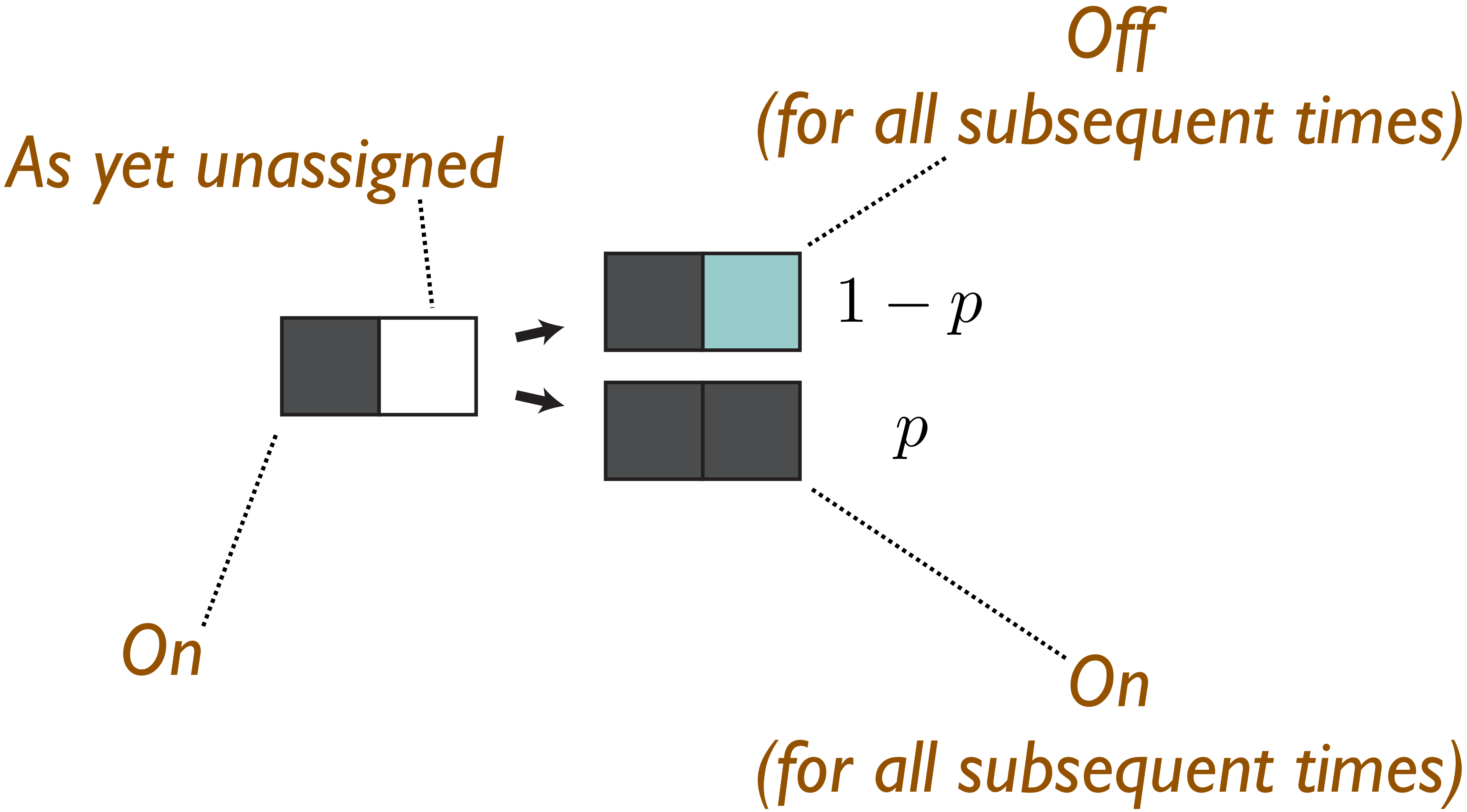


# Avalanche distribution



# Cluster growth model

- Apply update rule to unassigned cells that have activated neighbours



*Off*
*(for all subsequent times)*

*As yet unassigned*

$1 - p$

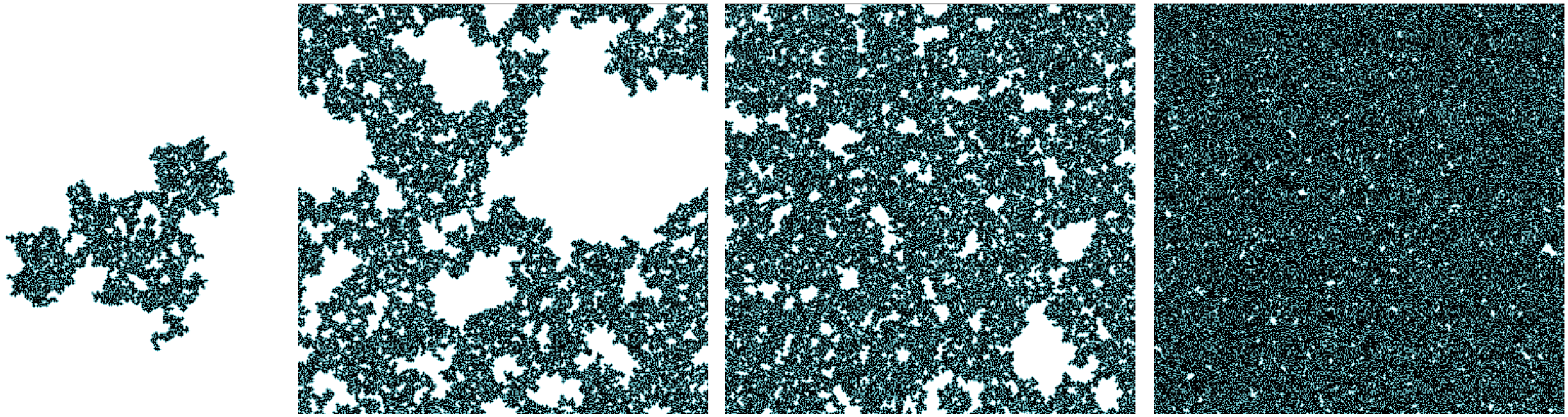
$p$

*On*

*On*
*(for all subsequent times)*

# Percolation transition

- Stationary final configurations on a 400x400 torus



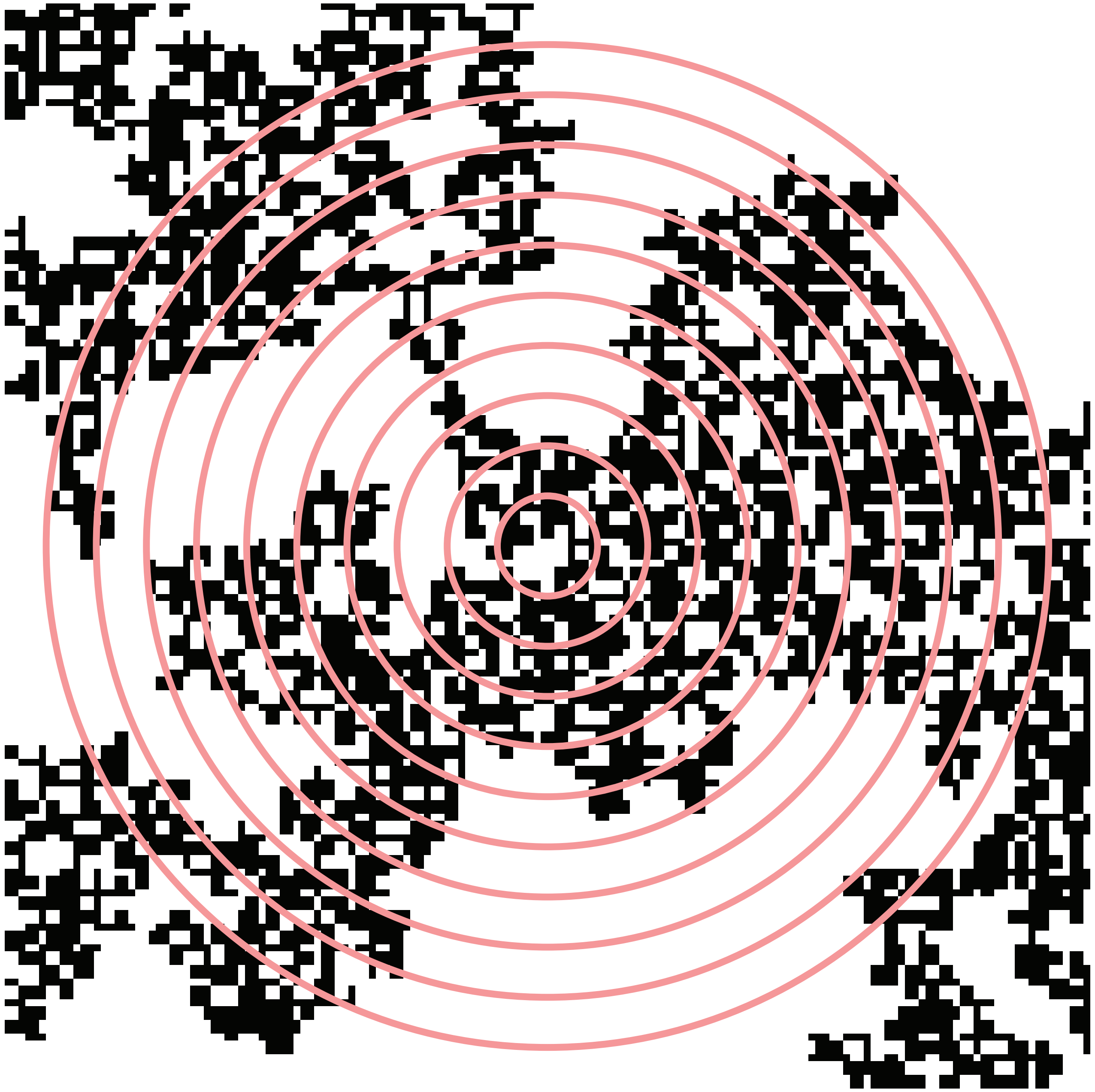
$p = 0.59$ $p_{\mathrm{c}} = 0.59274$ $p = 0.6$ $p = 0.7$

fractal

# Percolation transition

- Criticality here is not self-organized but instead requires fine tuning of the growth probability $p$
- Continuous phase transition at $p_\mathrm{c}$, where the clusters have fractal dimension intermediate between 1 and 2
- Scale invariance is lost at $p > p_\mathrm{c}$ and $p < p_\mathrm{c}$

# Fractal dimension



- Bin cell counts in circular shells
- Fit a linear slope on a log-log plot

$$N \sim R^{D_\mathrm{f}}$$

$$dN \sim R^{D_\mathrm{f}-1} dR$$

# Simple data structure

- 3-state grid of cells stored as conventional 2D array

```
const size_t halfL = 200;
const size_t L = 2*halfL;
enum cell_t { UNASSIGNED=2, ON=1, OFF=0 };
cell_t grid[L][L];

void initialize_grid(void)
{
   for (size_t i = 0; i < L; ++i)
      for (size_t j = 0; j < L; ++j)
         grid[i][j] = UNASSIGNED;
   grid[halfL][halfL] = ON;
}
```

# Simple updates

- Each sweep scales as $L^2$
- As many as $L$ sweeps for the On wavefront to propagate across the system

```
double prob;
void sweep_grid(void)
{
   for (size_t i = 0; i < L; ++i)
      for (size_t j = 0; j < L; ++j)
         if ( grid[i][j] == UNASSIGNED and
              (grid[(i+1)%L][j] == ON or
               grid[(L+i-1)%L][j] == ON or
               grid[i][(j+1)%L] == ON or
               grid[i][(L+j-1)%L] == ON) )
            grid[i][j] = (Rand() < prob ? ON : OFF);
}
```

# Redundant data structure

- Trade off memory for algorithmic efficiency

```
const size_t halfL = 200;
const size_t L = 2*halfL;
enum cell_t { UNASSIGNED = 2, ON = 1, OFF = 0 };
cell_t grid[L][L];

class coord
{ public:
   size_t x;
   size_t y;
   coord(size_t x_, size_t y_) : x(x_), y(y_) {}
};

#include <queue>
using std::queue;
queue<coord> perim;
```

# Initialize the perimeter

```
void initialize_grid(void)
{
   for (size_t i = 0; i < L; ++i)
      for (size_t j = 0; j < L; ++j)
         grid[i][j] = UNASSIGNED;
   grid[halfL][halfL] = ON;
   perim.push(coord(halfL+1,halfL));
   perim.push(coord(halfL-1,halfL));
   perim.push(coord(halfL,halfL+1));
   perim.push(coord(halfL,halfL-1));
}
```

# Grow the perimeter

```
double prob;
void sweep_grid(void)
{
   while (!perim.empty())
   {
      const coord c = perim.front(); perim.pop();
      const size_t i = c.x;
      const size_t j = c.y;
      if (grid[i][j] == UNASSIGNED)
         if (Rand() < prob)
         {
            grid[i][j] = ON;
            if (grid[(i+1)%L][j] == UNASSIGNED) perim.push(coord((i+1)%L,j));
            if (grid[(L+i-1)%L][j] == UNASSIGNED) perim.push(coord((L+i-1)%L,j));
            if (grid[i][(j+1)%L] == UNASSIGNED) perim.push(coord(i,(j+1)%L));
            if (grid[i][(L+j-1)%L] == UNASSIGNED) perim.push(coord(i,(L+j-1)%L));
          }
      else
         grid[i][j] = OFF;
   }
}
```