



# Computational Physics (PHYS6350)

*Lecture 20: Problems in statistical physics part II*

- Simulated annealing
- Percolation simulation

**April 6, 2023**

**Instructor:** Volodymyr Vovchenko ([vvovchenko@uh.edu](mailto:vvovchenko@uh.edu))

**Course materials:** <https://github.com/vlvovch/PHYS6350-ComputationalPhysics>

# Finding the ground state

---

Recall how in the Metropolis algorithm we move between states

For a move from state  $i$  to state  $j$

- If  $E_j < E_i$ , the move is unconditionally accepted.
- If  $E_j > E_i$ , the move is accepted with a probability

$$P_a = e^{-\beta(E_j - E_i)} \quad \beta = \frac{1}{T}$$

At  $T = 0$  only the single ground state  $E_i = 0$  contributes to the partition function, however, finding this state directly may be challenging (too long equilibration, have to probe a lot of states, can end up in a local minimum)

Alternative approach:

- Start at some finite  $T$
- Run the Metropolis algorithm to reach equilibrium
- Lower the temperature a bit and run the Metropolis algorithm to reach new equilibrium
- Repeat until  $T = 0$  where we will reach the ground state

# Simulated annealing

Apply this logic to find the global minimum of a function  $f(x)$

Helps avoid getting stuck in a local minimum

Example:  $f(x) = x^2 - \cos(4\pi x)$

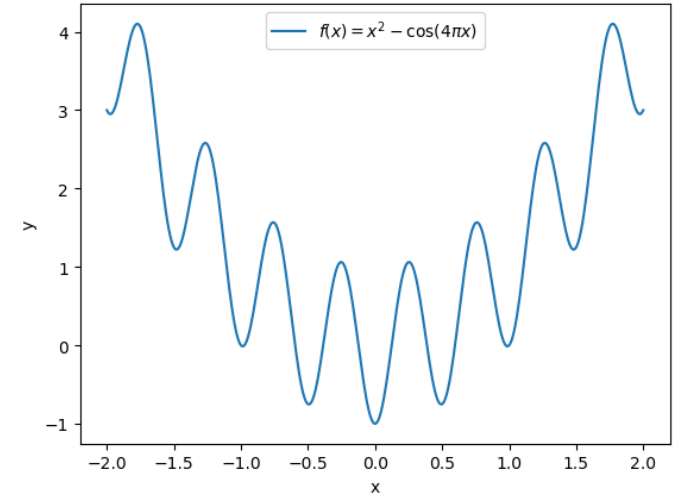
- Each step consider a move  $x \rightarrow x + \Delta x$  where  $\Delta x$  is drawn e.g. from the *normal distribution*
- Accept the move with *acceptance probability* a la Metropolis

$$p_a = e^{-[f(x_{\text{cand}}) - f(x)]/T}$$

- Lower the temperature  $T$  in the next step via a *cooling schedule*  
For example exponential schedule giving at step  $n$ :

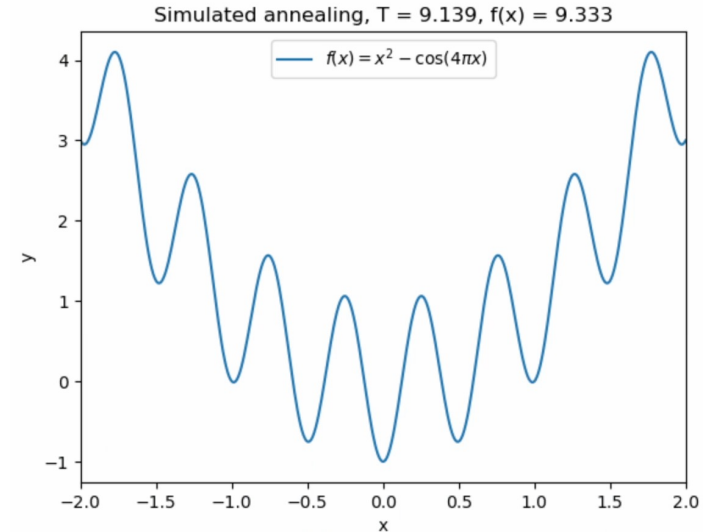
$$T_n = T_{\max} e^{-n/\tau}$$

- Stop the process once  $T$  goes below a certain threshold  $T < T_{\min}$



# Simulated annealing

```
# A single step of the simulated annealing process at temperature T
# for finding the minimum of a function f(x)
# The candidate move xcand is sampled from a Gaussian distribution
# around the present value of x, with a condition xmin <= xcand <= xmax
# The move is accepted with the Metropolis based probability
def simulated_annealing_step(f, T, x, sigma, xmin = -1e9, xmax = 1e9):
    fnow = f(x)
    xcand = np.random.normal(x, sigma)
    while (xcand < xmin or xcand > xmax):
        xcand = np.random.normal(x, sigma)
    fcand = f(xcand)
    if (fcand < fnow or (T > 0 and np.random.rand() < np.exp(-(fcand - fnow) / T))):
        return xcand
    else:
        return x
```



Apply it to our function

```
x = 2
sigma = 1.
Tmin = 1.e-3
Tmax = 10.
T = Tmax
t = 0
tau = 100.

while (T > Tmin):
    T = Tmax * np.exp(-t/tau)
    x = simulated_annealing_step(f, T, x, sigma)
    t += 1
```

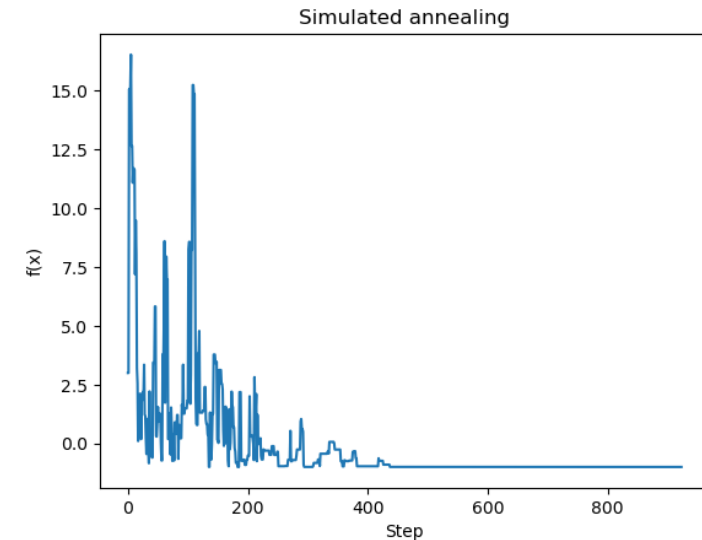
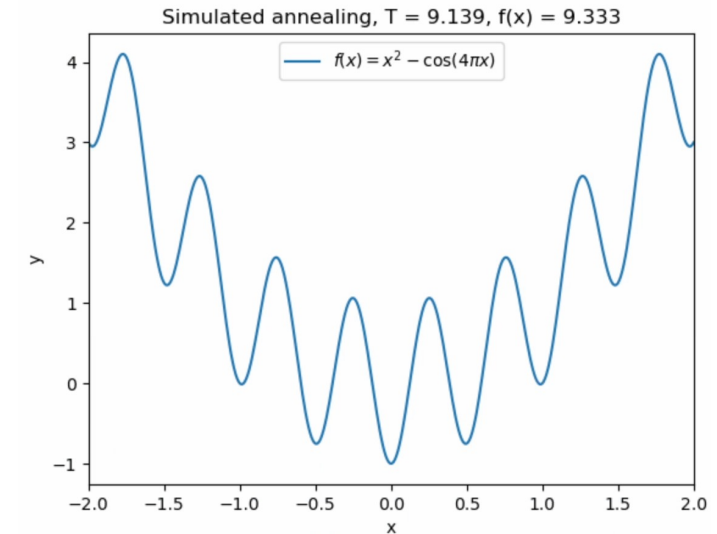
# Simulated annealing

```
# A single step of the simulated annealing process at temperature T
# for finding the minimum of a function f(x)
# The candidate move xcand is sampled from a Gaussian distribution
# around the present value of x, with a condition xmin <= xcand <= xmax
# The move is accepted with the Metropolis based probability
def simulated_annealing_step(f, T, x, sigma, xmin = -1e9, xmax = 1e9):
    fnow = f(x)
    xcand = np.random.normal(x, sigma)
    while (xcand < xmin or xcand > xmax):
        xcand = np.random.normal(x, sigma)
    fcand = f(xcand)
    if (fcand < fnow or (T > 0 and np.random.rand() < np.exp(-(fcand - fnow) / T))):
        return xcand
    else:
        return x
```

Apply it to our function

```
x = 2
sigma = 1.
Tmin = 1.e-3
Tmax = 10.
T = Tmax
t = 0
tau = 100.

while (T > Tmin):
    T = Tmax * np.exp(-t/tau)
    x = simulated_annealing_step(f, T, x, sigma)
    t += 1
```

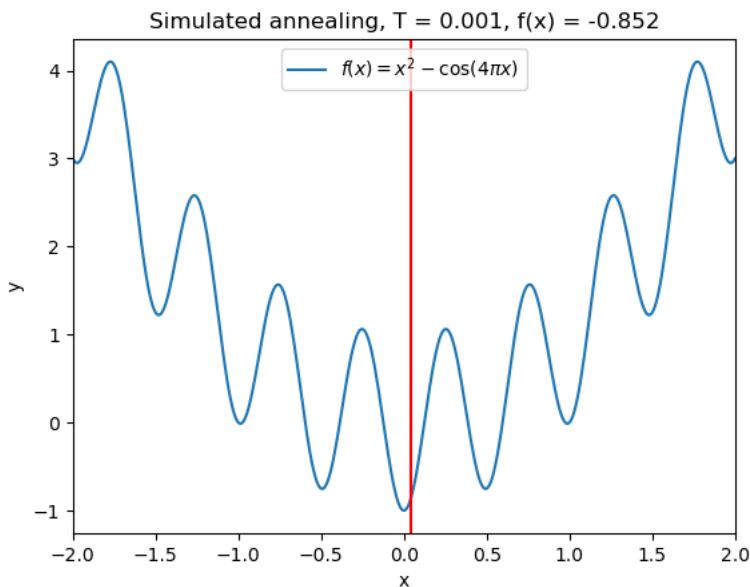


# Simulated annealing

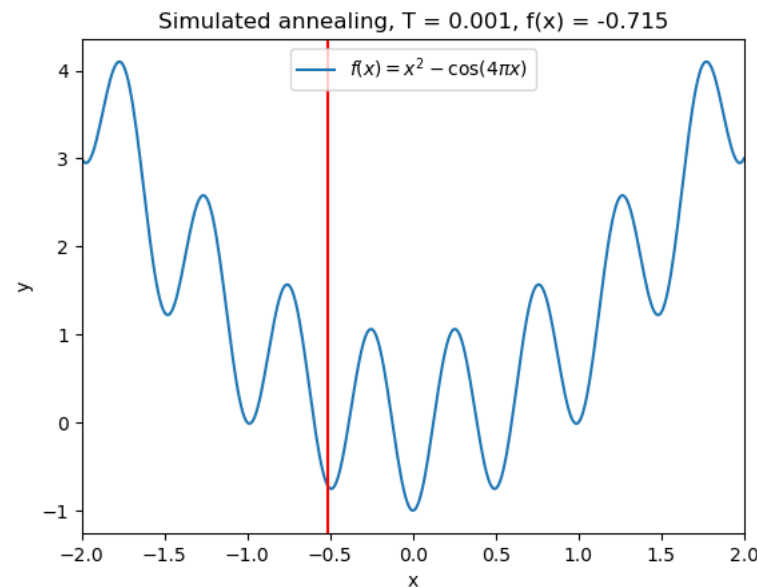
The solution is approximate and not guaranteed to give the global minimum, especially if cooling is too fast

E.g.  $\tau = 100 \rightarrow 10$

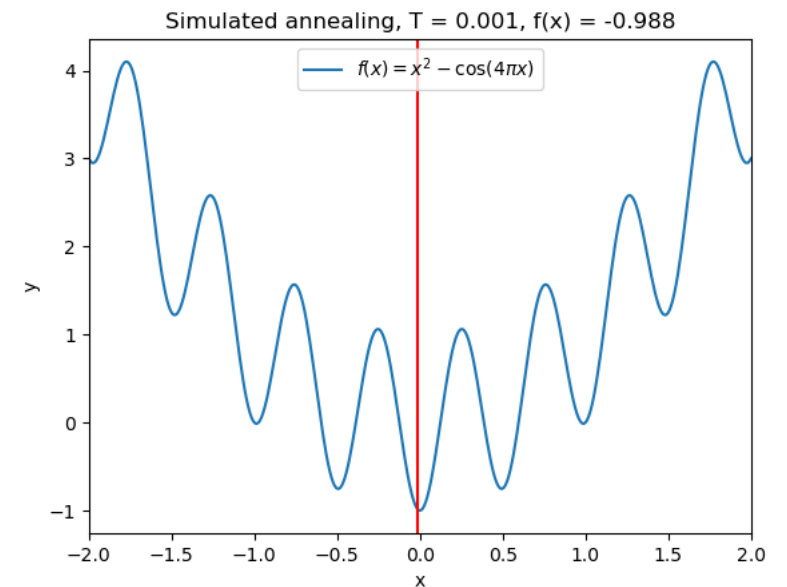
Run 1



Run 2



Run 3



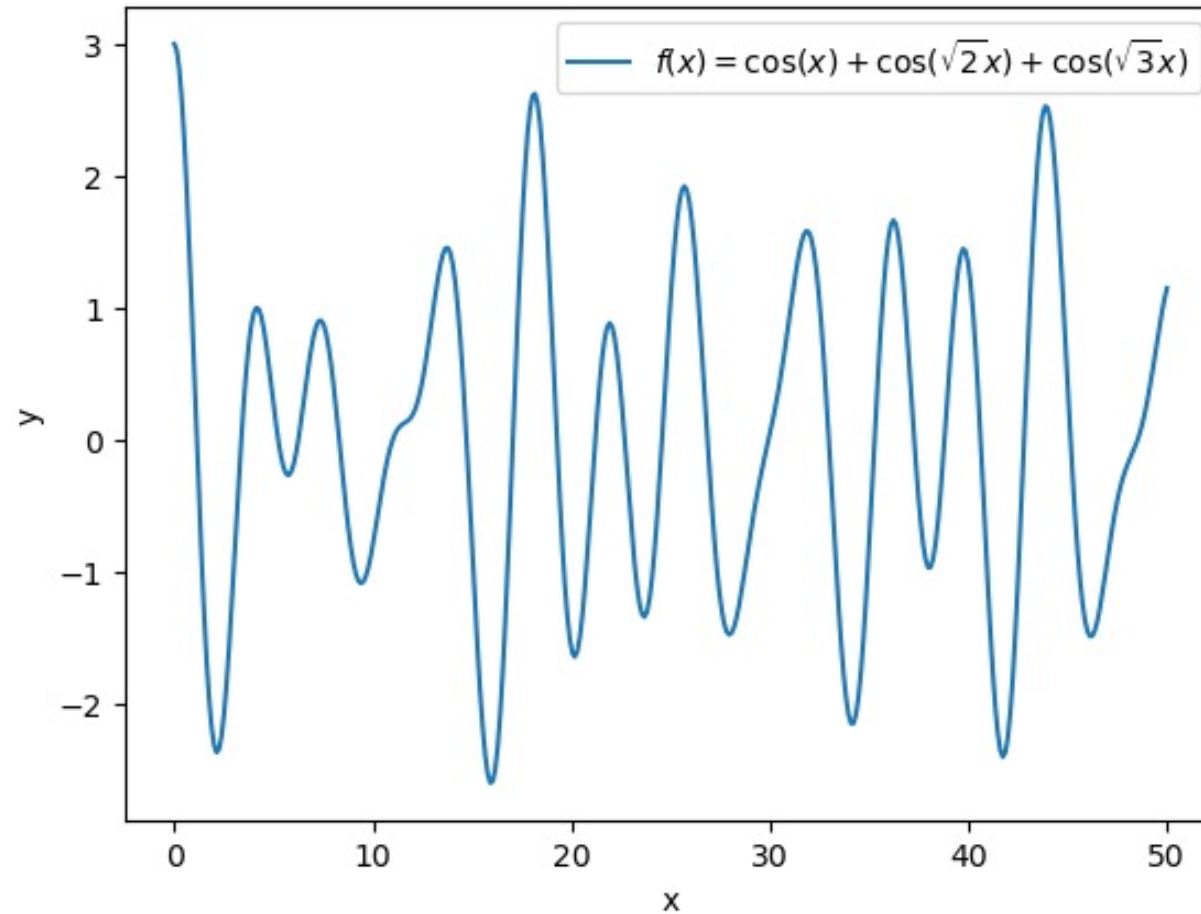
The choice of possible moves, acceptance probabilities, and cooling schedule should be varied and tailored for the problem at hand

# Simulated annealing

---

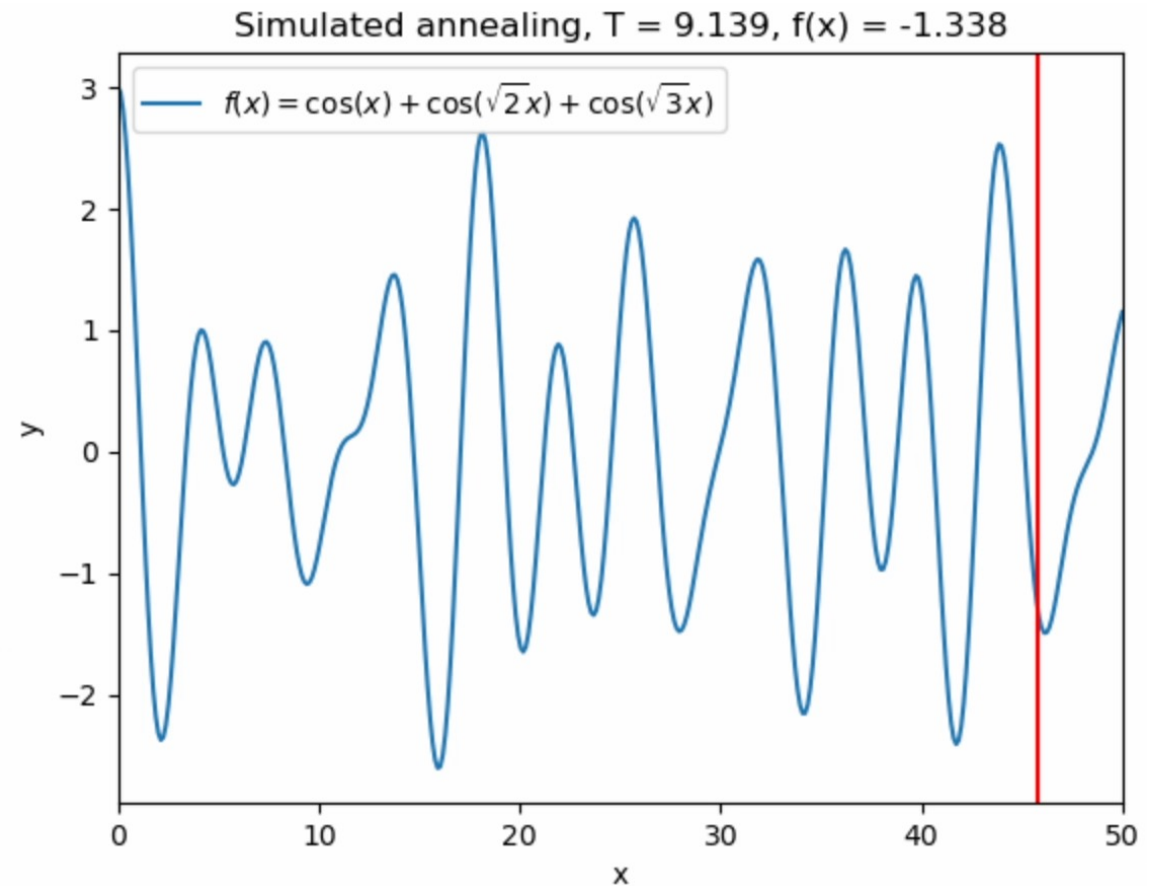
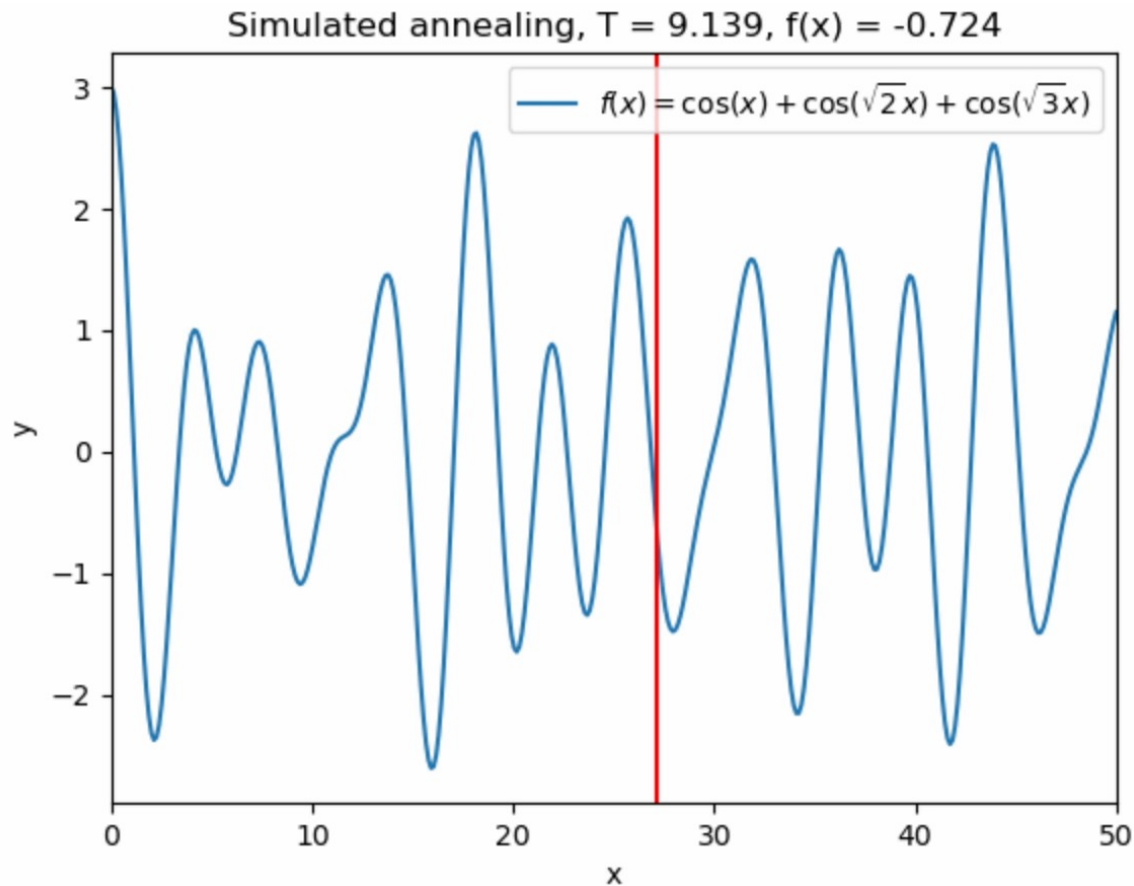
A more involved example

$$f(x) = \cos(x) + \cos(\sqrt{2}x) + \cos(\sqrt{3}x)$$



# Simulated annealing

A more involved example  $f(x) = \cos(x) + \cos(\sqrt{2}x) + \cos(\sqrt{3}x)$





# Simulated annealing and the traveling salesman problem

---

## Traveling salesman problem (TSP):

You have  $N$  cities where each pair of cities is connected by roads.  
You need to visit all cities exactly once starting ending the trip in the starting city.

The goal is to find a path with the shortest total distance to travel.

The traveling salesman problem is **NP-hard**

Known exact solutions are exponential in  $N$

Brute-force:  $O(N!)$

Held-Karp algorithm:  $O(N^2 2^N)$

Approximate solution with **simulated annealing**:

- Start with random path
- At each step swap two cities from the path and compute the total distance change  $\Delta D$
- Accept the new path with probability

$$P_a = \exp(-\Delta D/T)$$

- Decrease  $T$  in next step in accordance with annealing schedule

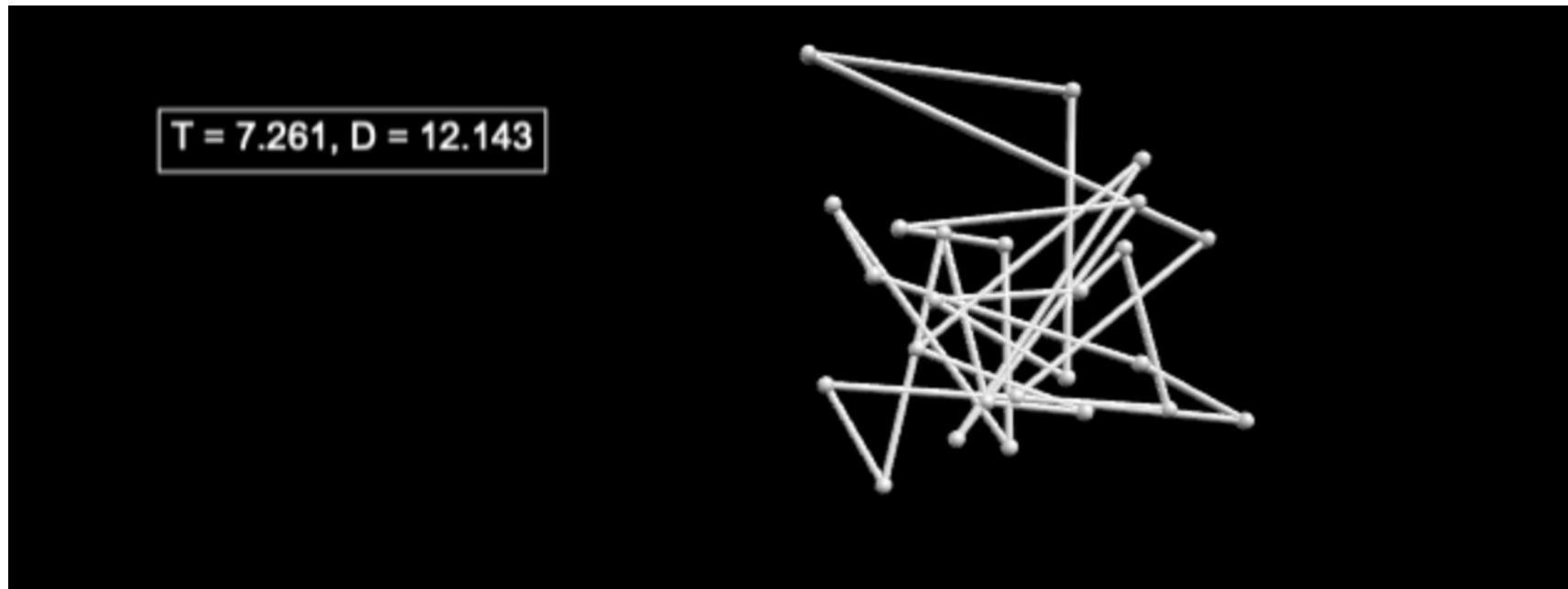


*Credit: Wikipedia*

# Simulated annealing and the traveling salesman problem

---

Code adapted from Example 10.4 from M. Newman, *Computational Physics*,  
<http://www-personal.umich.edu/~mejn/cp/programs/salesman.py>



# Percolation threshold

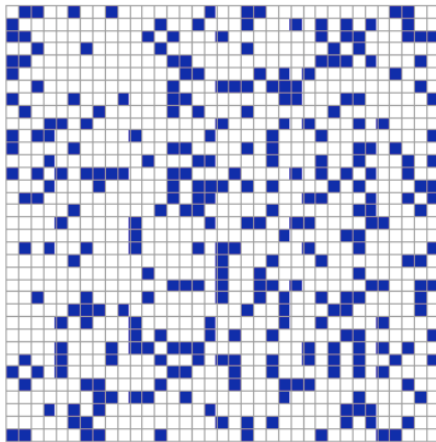
---

Percolation theory studies the formation of long-range connectivity in random systems. Usually, one can imagine a geometric configuration where certain “*conducting*” objects occupy a fraction of space

**Example:** square lattice

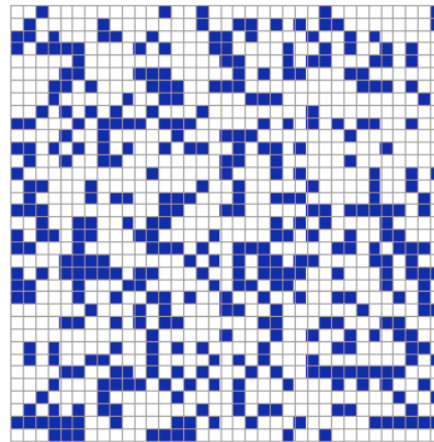
- Some fraction  $p$  of sites is occupied
- Occupied sites form clusters through connection to its four neighbors
- Above certain  $p_c$ , and long-range, “infinite” cluster forms

insulator



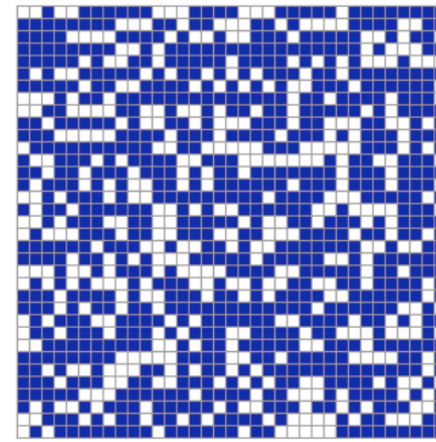
Site percolation with  $p = 0.25$

insulator



Site percolation with  $p = 0.35$

conductor

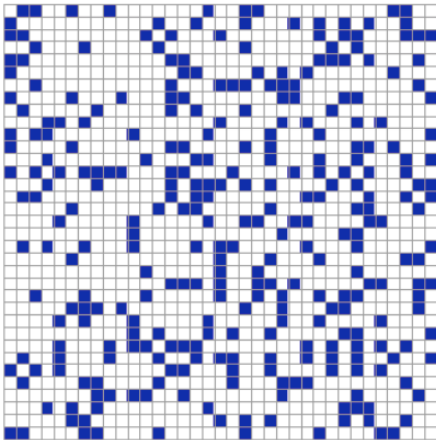


Site percolation with  $p = 0.65$

# Percolation threshold

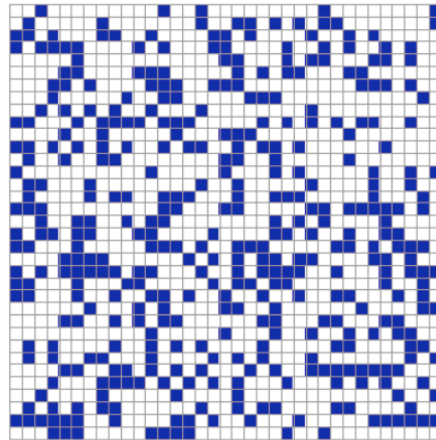
**Example:** square lattice

insulator



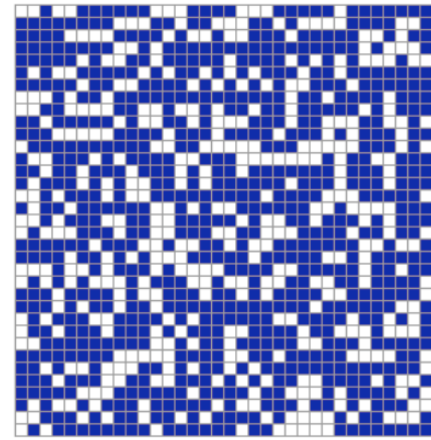
Site percolation with  $p = 0.25$

insulator



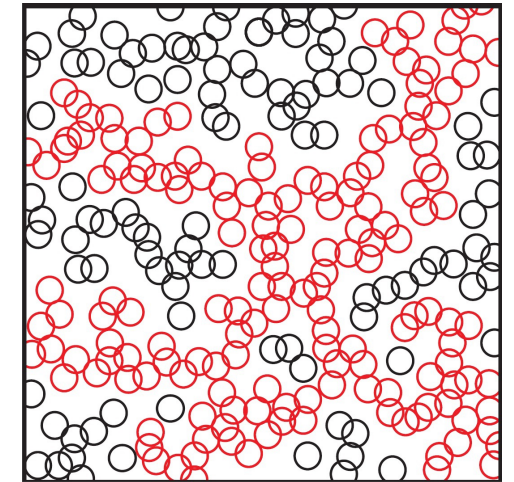
Site percolation with  $p = 0.35$

conductor



Site percolation with  $p = 0.65$

**Another example:**  
disk percolation



Images from <https://faculty.math.illinois.edu/~kkirkpat/percolation.html>

Above a **percolation threshold**  $p_c$  connectivity across the whole grid forms, insulator  $\rightarrow$  conductor

How to find  $p_c$ ?

# Percolation threshold simulation

---

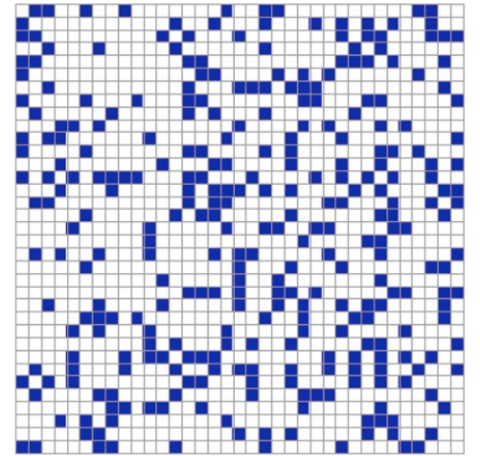
Simulate percolation threshold with Monte Carlo methods

## Strategy 1:

- For a given value of  $p$  mark each site as occupied with a probability  $p$
- Find all clusters (e.g. with [union-find data structure](#))
- Check if a conducting cluster from bottom to top edge exists
- We expect the cluster to exist for  $p > p_c$  and not exist for  $p < p_c$

## Strategy 2:

- Start with zero occupied site
- Mark a random unoccupied site as occupied and check if it forms a new cluster
- Repeat the process until a conducting cluster is formed
- The ratio of occupied to total sites is the estimate for  $p_c$



**Caveats:** finite-size effects

# Percolation threshold: union-find data structure

---

For finding clusters we can utilize the union-find data structure

```
# Class implementing the union-find structure with a 2D (x,y) index  
# This will also store the minimum and maximum y (vertical) coordinate of each set  
class UnionFind:  
    def __init__(self, N):  
        self.N = N  
        self.parent = list(range(N**2))  
        self.rank = [0] * N**2  
        self.maxy = [0] * N**2  
        self.miny = [0] * N**2  
        for x in range(N):  
            for y in range(N):  
                ind = self.index(x,y)  
                self.maxy[ind] = y  
                self.miny[ind] = y  
  
    def index(self, x, y):  
        return self.N * x + y  
  
    def findxy(self, x, y):  
        return self.find(self.index(x,y))  
  
    def find(self, ind):  
        if self.parent[ind] != ind:  
            self.parent[ind] = self.find(self.parent[ind])  
        return self.parent[ind]
```

```
    def unionxy(self, x1, y1, x2, y2):  
        self.union(self.index(x1,y1), self.index(x2,y2))  
  
    def union(self, ind1, ind2):  
        root_1 = self.find(ind1)  
        root_2 = self.find(ind2)  
  
        if root_1 == root_2:  
            return  
  
        if self.rank[root_1] > self.rank[root_2]:  
            self.maxy[root_1] = max(self.maxy[root_1], self.maxy[root_2])  
            self.miny[root_1] = min(self.miny[root_1], self.miny[root_2])  
            self.parent[root_2] = root_1  
        else:  
            self.maxy[root_2] = max(self.maxy[root_1], self.maxy[root_2])  
            self.miny[root_2] = min(self.miny[root_1], self.miny[root_2])  
            self.parent[root_1] = root_2  
            if self.rank[root_1] == self.rank[root_2]:  
                self.rank[root_2] += 1
```

Here implemented for a 2D grid, and keeps track of max and min y coordinate

# Percolation threshold: implementing strategy 2

---

```
## Simulate percolation threshold on a square in 2D
## Return the fraction of occupied cells at percolation
def simulateSquarePercolation(N):
    cells = np.zeros([N,N])
    occupied = 0
    uf = UnionFind(N)
    while True:
        # Choose unoccupied cell to occupy
        while True:
            ind = np.random.randint(N**2)
            x = ind // N
            y = ind % N
            if (cells[x][y] == 0):
                break
        cells[x][y] = 1
        occupied += 1

        # Add the new cell to existing clusters
        if (x > 0 and cells[x-1][y] == 1):
            uf.unionxy(x,y,x-1,y)
        if (x < N - 1 and cells[x+1][y] == 1):
            uf.unionxy(x,y,x+1,y)
        if (y > 0 and cells[x][y-1] == 1):
            uf.unionxy(x,y,x,y-1)
        if (y < N - 1 and cells[x][y+1] == 1):
            uf.unionxy(x,y,x,y+1)

    ymin = uf.miny[uf.find(ind)]
    ymax = uf.maxy[uf.find(ind)]

    # Check if percolating cluster is found
    if (ymin == 0 and ymax == N-1):
        return occupied / N**2
```



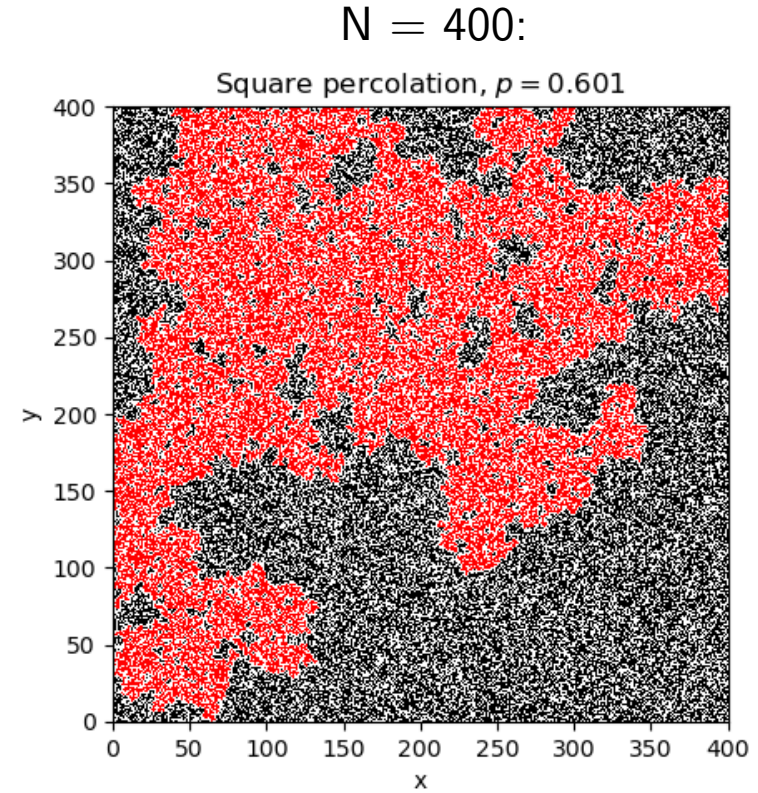
# Percolation threshold: implementing strategy 2

```
## Simulate percolation threshold on a square in 2D
## Return the fraction of occupied cells at percolation
def simulateSquarePercolation(N):
    cells = np.zeros([N,N])
    occupied = 0
    uf = UnionFind(N)
    while True:
        # Choose unoccupied cell to occupy
        while True:
            ind = np.random.randint(N**2)
            x = ind // N
            y = ind % N
            if (cells[x][y] == 0):
                break
        cells[x][y] = 1
        occupied += 1

        # Add the new cell to existing clusters
        if (x > 0 and cells[x-1][y] == 1):
            uf.unionxy(x,y,x-1,y)
        if (x < N - 1 and cells[x+1][y] == 1):
            uf.unionxy(x,y,x+1,y)
        if (y > 0 and cells[x][y-1] == 1):
            uf.unionxy(x,y,x,y-1)
        if (y < N - 1 and cells[x][y+1] == 1):
            uf.unionxy(x,y,x,y+1)

    ymin = uf.miny[uf.find(ind)]
    ymax = uf.maxy[uf.find(ind)]

    # Check if percolating cluster is found
    if (ymin == 0 and ymax == N-1):
        return occupied / N**2
```





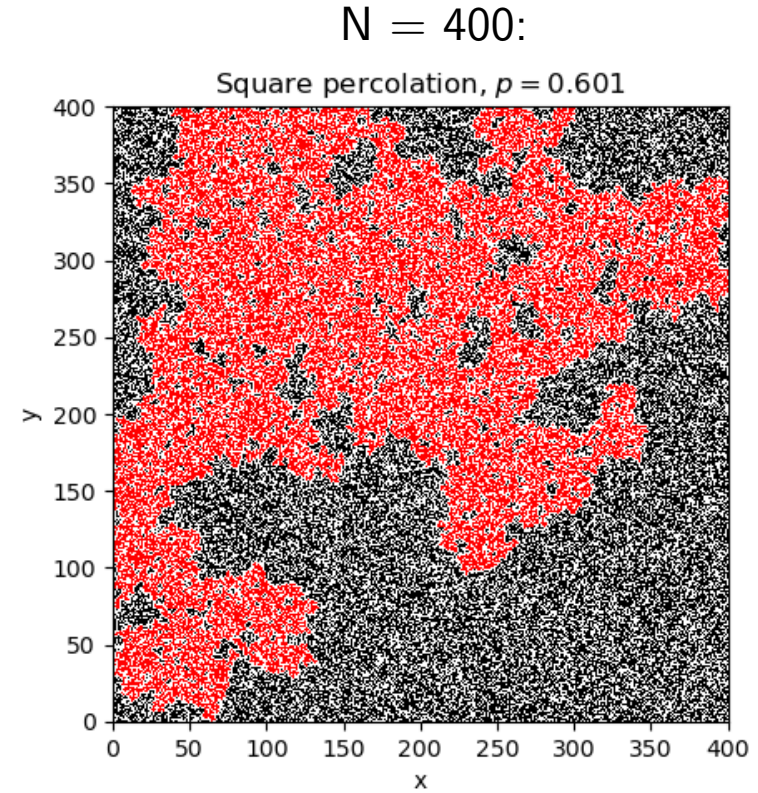
# Percolation threshold: implementing strategy 2

```
## Simulate percolation threshold on a square in 2D
## Return the fraction of occupied cells at percolation
def simulateSquarePercolation(N):
    cells = np.zeros([N,N])
    occupied = 0
    uf = UnionFind(N)
    while True:
        # Choose unoccupied cell to occupy
        while True:
            ind = np.random.randint(N**2)
            x = ind // N
            y = ind % N
            if (cells[x][y] == 0):
                break
        cells[x][y] = 1
        occupied += 1

        # Add the new cell to existing clusters
        if (x > 0 and cells[x-1][y] == 1):
            uf.unionxy(x,y,x-1,y)
        if (x < N - 1 and cells[x+1][y] == 1):
            uf.unionxy(x,y,x+1,y)
        if (y > 0 and cells[x][y-1] == 1):
            uf.unionxy(x,y,x,y-1)
        if (y < N - 1 and cells[x][y+1] == 1):
            uf.unionxy(x,y,x,y+1)

    ymin = uf.miny[uf.find(ind)]
    ymax = uf.maxy[uf.find(ind)]

    # Check if percolating cluster is found
    if (ymin == 0 and ymax == N-1):
        return occupied / N**2
```



10000 runs with N = 10:

Simulating square percolation

NxN = 10 x 10 with M = 10000 samples

p\_c = 0.5905830000000054 +- 0.023061418236939468

Literature value:

$p_c = 0.592746$