Percolation and Cluster Algorithms

Perhaps the simplest system which exhibits the characteristics of a sharp second order phase transition in the limit of infinite size is site or bond percolation on a two-dimensional lattice. The size of the system is the number of sites (or bonds) in the lattice. There is a single control variable, which is the probability $p$ that a lattice site is occupied.

This is not a thermodynamic system in the strict sense because there is no temperature defined. It can be viewed as a system with a variable number of particles that can occupy fixed lattice sites. If the temperature in the expression for the partition function is taken to be $T = \infty$, then the probability of any configuration is independent of its energy $E$. The factor $1/N!$ can be interpreted as implying that the particles are identical and indistinguishable: each distinct occupation pattern has the same probability.

Consider a square lattice with $N \times N$ sites. A site can be occupied or empty, so its state can be represented using a boolean variable, say true for occupied or false for empty.

Each interior lattice site has four nearest neighbors, one each to the north, south, east and west of it. Sites on the boundaries have fewer neighbors. A cluster on the lattice is a set of occupied sites mutually connected as nearest neighbors.

A percolating cluster or spanning cluster has at least one site on each of the four boundaries of the lattice.

The percolation problem assumes that each site can be occupied with constant probability $p$ in the range $[0, 1]$. For example, if $p = \frac{1}{2}$, then a coin toss can be used to generate a typical sample.

There is a critical probability $p_c$ below which spanning clusters are extremely rare and above which
they become increasingly common. As $N \to \infty$ this transition becomes sharp, like the phase transition from a ferromagnet to a paramagnet at the Curie temperature $T_c$.

**Scaling and Critical Exponents**

A second order phase transition has characteristic singular behavior in the limit of infinite size. These singularities appear in the equations of state at the critical value of the control parameter, and are characterized by numbers called critical exponents. These critical exponents are very useful and interesting because they appear to take the numerical same values for systems with the same dimensionality and symmetries, and are independent of the microscopic details of individual systems.

**Order Parameter**

A convenient order parameter for the percolation problem is defined to be the fraction of occupied sites in the spanning cluster

$$F(p) \equiv \frac{\text{Number of sites in spanning cluster}}{\text{Total number of occupied sites}}.$$ 

In the infinite size limit, this order parameter has the singular behavior

$$F(p) = \begin{cases} 
0 & \text{for } p < p_c \\
\text{const} \times (p - p_c)^\beta & \text{for } p \geq p_c 
\end{cases},$$

where the order parameter critical exponent $\beta = 5/36$ for percolation on a 2-D square lattice.

**Cluster Size Distribution**
The cluster size distribution of a lattice configuration is defined to be

\[ n_s(p) = \frac{\text{Number of clusters with } s \text{ occupied sites}}{\text{Number of sites in the lattice}}. \]

By convention, spanning clusters are not included in calculating this distribution. The probability that an occupied site belongs to a cluster of \( s \) sites is

\[ w_s(p) = \frac{sn_s}{\sum_s sn_s}. \]

**Average Cluster Size**

The average or mean cluster size in a lattice configuration is defined as

\[ S(p) = \sum_s sw_s = \frac{\sum_s s^2n_s}{\sum_s sn_s}. \]

Here again, spanning clusters are excluded from this calculation so the average cluster size is finite above the percolation threshold. In the limit of infinite lattice size the average cluster size nevertheless becomes singular near the percolation threshold

\[ S(p) \sim \frac{1}{|p - p_c|^\gamma}, \]

where the cluster size critical exponent \( \gamma = 43/18 \) for percolation on a 2-D square lattice.
**Correlation Length**

Two sites on the lattice are said to be correlated or connected if they are both occupied and belong to the same cluster. The correlation length $\xi$ of a lattice configuration is the average distance between connected sites. By convention, spannings cluster are not included in calculating the correlation length so that it remains finite in the limit of infinite lattice size. Even so, the correlation length diverges near the percolation phase transition

$$\xi(p) \sim \frac{1}{|p - p_c|^\nu},$$

where the correlation length critical exponent $\nu = 4/3$ for percolation on a 2-D square lattice.

**Scaling Laws**

There are theoretical and experimental indications that the critical exponents that characterize second-order phase transitions are not all independent, but are connected by scaling laws. The scaling law which appears to apply to systems with transitions similar to percolation is

$$2\beta + \gamma = \nu d,$$

where $d$ is the spatial dimensionality of the system. For percolation in $d = 2$ dimensions, the theoretical predictions indeed satisfy this scaling relation:

$$2 \times \frac{5}{36} + \frac{43}{18} = \frac{4}{3} \times 2.$$
Finite-Size Scaling Theory

Sharp phase transitions, critical singularities and exponents, and scaling laws are all properties of infinitely large systems. Experiments and computer simulations deal with finite systems. To measure or simulate properties that hold in the thermodynamic limit, one can study a sequence of systems of ever increasing size and extrapolate measured properties to the limit of infinite size. Finite-size scaling theory provides predictions for scaling behavior in this limit.

The basic concept that underlies finite-size scaling theory involves the correlation length $\xi$ of the system. In an infinite system, the correlation length diverges at the critical point. In a finite system of linear size $L$, the correlation length is of course limited to $\xi \leq L$. In practice, there will be a small range of values of the control parameter, the occupation probability $p$ in the case of percolation, near which the measured correlation length $\xi(p)$ diverges as

$$\xi(p) \sim L \sim \frac{1}{|p - p_c(L)|^\nu}, \quad \lim_{L \to \infty} p_c(L) = p_c,$$

where $p_c(L)$ tends to the critical point $p_c$ of the infinite system. This approximate equation can be solved for

$$|p - p_c| \sim \frac{1}{L^{1/\nu}}.$$

The singular behavior of the order parameter and average cluster size for finite systems of increasing $L$ can be estimated as

$$F(p) \sim (p - p_c)^\beta \sim L^{-\beta/\nu},$$

$$S(p) \sim |p - p_c|^{-\gamma} \sim L^{\gamma/\nu}.$$
The ratios $\beta/\nu$ and $\gamma/\nu$ extracted from measurements of $F(p_c)$ and $S(p_c)$ as functions of system size $L$. 
Monte Carlo Simulation of Percolation

The percolation problem is a very interesting application of Monte Carlo methods and random numbers to study the properties of a phase transition.

The lattice has $2^{N^2}$ possible states. If $N = 20$, $2^{20^2} = 2.6 \times 10^{120}$. This is an astronomical number: if we tried to enumerate all states at the rate of say 1,000,000 per second on a fast computer, it would take $8.2 \times 10^{106}$ years to complete the task!

Short of obtaining an exact analytical solution, the best way to solve this type of problem is to sample configurations at random! A numerical experiment consists in generating a large number of such samples, and then measuring average properties of the system.

We need to identify all clusters in the sample and label them say 1,2,3,... Each occupied site is labeled with the number of the cluster to which it belongs.

### C++ Percolation Code

```cpp
int N = 20; // number of sites in x and y
double p = 0.55; // site occupation probability
bool** occupied; // NxN array to store the state of the lattice

int clusters; // number of clusters
int** label; // NxN array of cluster labels for each site
int currentLabel; // label of current cluster
```
int spanningLabel; // label of spanning cluster

void addNewNeighbor(int i, int j) {
    if (occupied[i][j] && // site is occupied, and
        label[i][j] == 0) // not yet labeled
    {
        label[i][j] = currentLabel;
        if (i < N-1)
            addNewNeighbor(i+1, j); // east
        if (i > 0)
            addNewNeighbor(i-1, j); // west
        if (j < N-1)
            addNewNeighbor(i, j+1); // north
        if (j > 0)
            addNewNeighbor(i, j-1); // south
    }
}

void newSample() {

    // visit each site and occupy it with probability p
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            occupied[i][j] = (rand() / (RAND_MAX + 1.0)) < p;
        }
    }
}
label[i][j] = 0;  // not yet labeled
}

// find and label all clusters of occupied sites
clusters = 0;
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        if (occupied[i][j] && // site is occupied, and
            label[i][j] == 0) // not yet labeled
        {
            currentLabel = ++clusters; // assign a new label
            addNewNeighbor(i, j);       // add to new cluster
        }
    }
}

// check each cluster for percolation
spanningLabel = 0;
for (int cluster = 1; cluster <= clusters; ++cluster) {
    // check west boundary sites
    bool west = false;
    for (int j = 0; j < N; j++)
if (label[0][j] == cluster) { west = true; break; }

// check east boundary sites
bool east = false;
for (int j = 0; j < N; j++)
    if (label[N-1][j] == cluster) { east = true; break; }

// check south boundary sites
bool south = false;
for (int i = 0; i < N; i++)
    if (label[i][0] == cluster) { south = true; break; }

// check north boundary sites
bool north = false;
for (int i = 0; i < N; i++)
    if (label[i][N-1] == cluster) { north = true; break; }

// check whether cluster touches all boundaries
if (west && east && south && north) {
    spanningLabel = cluster;
    break;
}
}
void initialize() {
    // allocate memory for arrays
    occupied = new bool* [N];
    label = new int* [N];
    for (int i = 0; i < N; i++) {
        occupied[i] = new bool [N];
        label[i] = new int [N];
    }
    newSample();
}
Cluster Algorithms to Reduce Critical Slowing Down

Monte Carlo simulations close to a phase transition are affected by critical slowing down. In the 2-D Ising system, the correlation length $\xi$ becomes very large, and the correlation time $\tau$, which measures the number of steps between independent Monte Carlo configurations behaves like

$$\tau \sim \xi^z,$$

where the dynamic critical exponent $z \simeq 2.1$ for the Metropolis algorithm.

The maximum possible value for $\xi$ in a finite system of $N = L \times L$ spins is $\xi \sim L$, because $\xi$ cannot be larger than the lattice size! This implies that $\tau \sim L^{2.1} \sim N$. This makes simulations difficult because the Metropolis algorithm time scales like $N$, so the time to generate independent Metropolis configurations scales like $N\tau \sim N^2 = L^4$. If the lattice size $L \to \sqrt{10}L \simeq 3.2L$, the simulation time increases by a factor of 100.

There is a simple physical argument which helps understand why $z = 2$: The Metropolis algorithm is a local algorithm, i.e., one spin is tested and flipped at a time. Near $T_c$ the system develops large domains of correlated spins which are difficult to break up. So the most likely change in configuration is the movement of a whole domain of spins. But one Metropolis sweep of the lattice can move a domain at most by approximately one lattice spacing in each time step. This motion is stochastic, i.e., like a random walk. The distance traveled in a random walk scales like $\sqrt{\text{time}}$, so to move a domain a distance of order $\xi$ takes $\tau \sim \xi^2$ Monte Carlo steps.

This argument suggests that the way to speed up a Monte Carlo simulation near $T_c$ is to use a non-local algorithm.
Swendsen-Wang Cluster Algorithm

The essential idea of this algorithm suggested by R.H. Swendsen and J.-S. Wang, *Phys. Rev. Lett.* 58, 86 (1987), is to identify clusters of like spins and treat each cluster as a giant spin to be flipped according to a random criterion. It is necessary that the algorithm obey the detailed balance condition. Swendsen and Wang found the following algorithm based on ideas from *percolation theory*:

**Freeze/delete bonds:** The 2-D square lattice, periodic boundary conditions, has $N = L \times L$ spins and $2N$ bonds between spins. Construct a *bond lattice* as follows:

- If the bond connects opposite spins, then delete it, i.e., temporarily uncouple the two spins. Note that opposite spins have a higher bond energy $+J$ if $J > 0$ and thus a higher effective temperature. So if $J$ is large we are effectively “melting” the bond.

- If the bond connects like spins (both up or both down), then delete the bond with probability $e^{-2J/(k_B T)}$, i.e., generate a random deviate $r$ and delete the bond if $r < e^{-2J/(k_B T)}$. Note that a like-spin pair has bond energy $-J$: so the change in energy in flipping one spin of the pair, i.e., in going from like to unlike spins is $\Delta E = 2J$. Bonds which survive this test are “frozen”. The probability of this happening is $1 - e^{-2J/(k_B T)}$. If $T = 0$ all like-spin bonds get frozen, while at $T = \infty$ the freezing probability is zero and all the bonds melt.

Note that constructing the bond lattice takes time of $\mathcal{O}(N)$ because there are $2N$ bonds.

**Cluster Decomposition:** After the bond lattice has been set up, the spins are decomposed into clusters. A cluster is simply a domain of spins connected to one another by frozen bonds. The lattice obviously decomposes into clusters in a unique way, and the decomposition is a deterministic problem.

Cluster decomposition is potentially time consuming. A naive algorithm can take time of $\mathcal{O}(N^2)$, so
it is essential to use a decomposition algorithm that scales linearly with lattice size like Metropolis!

**Spin Update:** So far, constructing the bond lattice and identifying clusters has not changed any of the spins. The spins in each cluster are now “frozen” and the bonds between different clusters have been deleted. Each cluster is now updated by assigning a random new value \( \pm 1 \) to all of the spins simultaneously, i.e., generate a random deviate \( r \) and flip all spins in that cluster if \( r < 0.5 \). Note that \( T \) does not play a role in this flipping decision.

The spin update step scales like the number of clusters which is \( < N \).

Swendsen and Wang showed that \( z \approx 0.35 \) for this algorithm in the 2-D Ising model. Assuming that each Swendsen-Wang step scales like \( N \), the running time for the simulation scales like

\[
N \tau \sim N \xi^{0.35} \sim NL^{0.35} \sim N^{1.175},
\]

which is *much* better than \( \mathcal{O}(N^2) \) with Metropolis.

The figure above from their paper shows a plot of the correlation time for the energy-energy correlation function, determined from the exponential behavior of the long-time tail, as a function of the linear lattice size.

The data show that their cluster algorithm becomes more efficient than the standard Metropolis algorithm for lattices sizes larger than \( \sim 18 \times 18 \).
FIG. 1. Log-log plots of correlation times for Monte Carlo simulations of the two-dimensional Ising model at the critical temperature as a function of the linear dimension $L$. The circles show data for a standard Monte Carlo simulation, and the line marked “$\tau \propto L^{2.125}$” gives the expected asymptotic slope (Ref. 4). The crosses show data for the new method, with a least-squares fit labeled with its slope of “$\tau \propto L^{0.35}$.”
Wolff Single Cluster Algorithm


The above table from Wolff’s paper shows results on the 2-D Ising model $n = 1$ on the first three rows, the $x - y$ model $n = 2$, and the classical Heisenberg model $n = 3$.

For the Ising model, the simulations are run at the critical temperature $T_c = 2.269$, $\beta = 1/T = 1/2.269 = 0.4406$. The correlation time is measured for the magnetic susceptibility which he defines as

$$ \chi = \frac{1}{L^2} \left\langle \left[ \sum_{i=1}^{L^2} \sigma_i \right]^2 \right\rangle. $$

This is essentially the same as the conventionally defined isothermal susceptibility per spin

$$ \chi = \left( \frac{\partial \langle m \rangle}{\partial H} \right)_T = \left( \frac{\partial \langle \sum_i s_i \rangle}{\partial H} \right)_T $$

$$ = \frac{1}{k_B T} \left[ \left\langle \frac{1}{N} \left( \sum_{i=1}^{N} s_i \right)^2 \right\rangle - \left( \frac{1}{N} \left\langle \sum_i s_i \right\rangle \right)^2 \right]. $$

If $H = 0$, then $\langle \sum s_i \rangle = 0$ by symmetry for a finite-sized system, so the two definitions agree up to a multiplicative factor $k_B T$. 
TABLE I. Results for the magnetic susceptibilities $\chi$ and autocorrelation times $\tau_\chi$ (in units comparable to sweeps) for simulations of $O(n)\sigma$ models on $L^2$ lattices. In each run a total of $c$ update steps have been performed involving clusters of an average of $\langle |c| \rangle$ spins. The effective autocorrelation time $\tau_\chi^{\text{eff}}$ is directly relevant for error estimation.

| $n$ | $L$ | $\beta$ | $c \times 10^{-6}$ | $\langle |c| \rangle L^{-2}$ | $\chi L^{-2}$ | $\tau_\chi$ | $\tau_\chi^{\text{eff}}$ |
|-----|-----|---------|-----------------|------------------|----------------|------------|----------------|
| 1   | 32  | 0.4406... | 0.50           | 0.4602(7)        | 0.4598(7)      | 2.3(3)     | 1.4           |
| 1   | 64  | 0.4406... | 0.25           | 0.3858(11)       | 0.3852(10)     | 2.3(3)     | 1.9           |
| 1   | 128 | 0.4406... | 0.20           | 0.3225(11)       | 0.3229(10)     | 2.7(5)     | 1.8           |
| 2   | 32  | 1.12      | 0.62           | 0.3582(5)        | 0.4420(2)      | 3.6(7)     | 1.4(2)        |
| 2   | 64  | 1.12      | 0.26           | 0.3043(6)        | 0.3754(3)      | 2.4(6)     | 1.3(2)        |
| 2   | 128 | 1.12      | 0.20           | 0.2582(7)        | 0.3190(3)      | 2.1(6)     | 1.2(2)        |
| 2   | 32  | 1.07      | 0.26           | 0.3247(7)        | 0.3985(4)      | 1.5(3)     | 1.5(3)        |
| 2   | 64  | 1.07      | 0.13           | 0.2629(9)        | 0.3245(5)      | 1.1(2)     | 1.1(2)        |
| 2   | 128 | 1.07      | 0.10           | 0.2114(9)        | 0.2608(5)      | 1.2(2)     | 1.2(2)        |
| 2   | 128 | 1.04      | 0.26           | 0.1638(5)        | 0.2032(5)      |            | 1.4(2)        |

$\langle |c| \rangle$, $\chi$
The Wolff algorithm works as follows:

- Choose a spin $s_i$ at random in the lattice flip it and mark it as a cluster spin.
- Grow a cluster with this spin as “seed” by checking each of its 4 neighbor spins:
  - If the neighbor is marked as a cluster spin, continue with the next neighbor, or quit if 4 neighbors are done. Otherwise:
  - If this neighbor is opposite to the seed spin, then add it to the cluster with probability $1 - e^{-2J/(k_BT)}$, i.e., generate a uniform deviate $r$ and if $r < 1 - e^{-2J/(k_BT)}$ flip the spin, mark it as belonging to the cluster, and recursively check each of its 4 neighbor spins.

Note that the decision to add a spin to the cluster follows the same probability criterion $r < 1 - e^{-2J/(k_BT)}$ to freeze a bond in the Swendsen-Wang algorithm. This implies that Wolff clusters have the same statistical properties as Swendsen-Wang clusters.

A simple argument shows that the Wolff algorithm is potentially more efficient than the Swendsen-Wang algorithm. Imagine the lattice partitioned into Swendsen-Wang clusters. The Wolff algorithm flips a single cluster got by choosing the seed site at random. This random choice obviously favors larger clusters. Flipping larger clusters is more likely to result in uncorrelated configurations!
Code for the Wolff Cluster Algorithm

The following program `wolff.cpp` codes the Wolff cluster algorithm for the 2-D Ising model. Following the suggestions in Wolff’s paper, the magnetic susceptibility per spin $\chi$, and the autocorrelation time $\tau_\chi$ for this observable are measured at the critical temperature $T_c = 2 / \log(1 + \sqrt{2}) = 2.2691853...$ of the infinite system.

```cpp
// Wolff cluster algorithm for the 2-D Ising Model

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <list>
using namespace std;

inline double std_rand()
{
    return rand() / (RAND_MAX + 1.0);
}

double J = +1; // ferromagnetic coupling
int Lx, Ly; // number of spins in x and y
int N; // number of spins
```
Variables for the cluster algorithm

The Wolff algorithm works by choosing a spin at random and then constructing one cluster of like spins by examining neighboring bonds and freezing them with probability

\[ 1 - e^{-2J/(k_B T)} . \]

We will use an \( L_x \times L_y \) array of bools called \texttt{cluster} to mark whether a spin belongs to the cluster or not.
bool **cluster; // cluster[i][j] = true if i,j belongs
double addProbability; // 1 - e^(-2J/kT)

void initializeClusterVariables() {

    // allocate 2-D array for spin cluster labels
    cluster = new bool* [Lx];
    for (int i = 0; i < Lx; i++)
        cluster[i] = new bool [Ly];

    // compute the probability to add a like spin to the cluster
    addProbability = 1 - exp(-2*J/T);
}

**One Wolff Monte Carlo step**

The Wolff algorithm is much simpler than the Swendsen-Wang algorithm because the lattice does not need to be partitioned into clusters. At each Monte Carlo step, a single cluster is grown around a randomly chosen seed spin, and all of the spins in this cluster are flipped.

    // declare functions to implement Wolff algorithm
    void growCluster(int i, int j, int clusterSpin);
void tryAdd(int i, int j, int clusterSpin);

void oneMonteCarloStep() {

    // no cluster defined so clear the cluster array
    for (int i = 0; i < Lx; i++)
        for (int j = 0; j < Lx; j++)
            cluster[i][j] = false;

    // choose a random spin and grow a cluster
    int i = int(std_rand() * Lx);
    int j = int(std_rand() * Ly);
    growCluster(i, j, s[i][j]);

    ++steps;
}

Growing a Wolff cluster

The following function grows a Wolff cluster and simultaneously flips all of the spins in the cluster. This is done in two simple steps:

- First the spin is marked as belonging to the cluster, and the spin is also flipped.
- Next, the four nearest neighbors as visited: if the neighbor does not already belong to the cluster,
then an attempt is made to add it by calling the tryAdd function. The variable clusterSpin holds the value $(\pm 1)$ of the seed spin. We will see further below that the tryAdd function call growCluster on the neighbor spin if it succeeds: thus the two functions call one another recursively until the growth stops.

```cpp
void growCluster(int i, int j, int clusterSpin) {

    // mark the spin as belonging to the cluster and flip it
    cluster[i][j] = true;
    s[i][j] = -s[i][j];

    // find the indices of the 4 neighbors
    // assuming periodic boundary conditions
    int iPrev = i == 0 ? Lx-1 : i-1;
    int iNext = i == Lx-1 ? 0 : i+1;
    int jPrev = j == 0 ? Ly-1 : j-1;
    int jNext = j == Ly-1 ? 0 : j+1;

    // if the neighbor spin does not belong to the
    // cluster, then try to add it to the cluster
    if (!cluster[iPrev][j])
        tryAdd(iPrev, j, clusterSpin);
    if (!cluster[iNext][j])
        tryAdd(iNext, j, clusterSpin);
```

wolff.cpp
tryAdd(iNext, j, clusterSpin);
if (!cluster[i][jPrev])
    tryAdd(i, jPrev, clusterSpin);
if (!cluster[i][jNext])
    tryAdd(i, jNext, clusterSpin);
}

Next, we define the function tryAdd which tests whether or not to add a candidate spin $s_{ij}$ to the cluster based on a Boltzmann criterion. The variable clusterSpin holds the value ($\pm 1$) of the seed spin. The candidate spin is added if

1. $s_{ij} = s_{\text{seed}}$, and
2. a random deviate is $< 1 - e^{-2J/(k_B T)}$.

```cpp
void tryAdd(int i, int j, int clusterSpin) {
    if (s[i][j] == clusterSpin)
        if (std_rand() < addProbability)
            growCluster(i, j, clusterSpin);
}
```

If the tests are successful, then tryAdd calls growCluster on the candidate spin $s_{ij}$.

**Measuring observables**
Next, we define variables and functions to measure various observables during the simulation. To reproduce the results in Wolff’s paper, we need to measure

- the susceptibility \( \chi \),
- the auto-correlation time of susceptibility measurements,
- and the error in the average susceptibility measured in two ways:
  - using the Monte Carlo error estimate, and
  - measuring the fluctuations in blocks of 1000 measurements.

```cpp
// variables to measure chi and its error estimate
double chi; // current susceptibility per spin
double chiSum; // accumulate chi values
double chiSqdSum; // accumulate chi^2 values
int nChi; // number of values accumulated

// variables to measure autocorrelation time
int nSave = 10; // number of values to save
double cChiSum; // accumulate
list<double> chiSave; // the saved values
double *cChi; // correlation sums
int nCorr; // number of values accumulated
```
// variables to estimate fluctuations by blocking
int stepsPerBlock = 1000; // suggested in Wolff paper
double chiBlock; // used to calculate block average
double chiBlockSum; // accumulate block <chi> values
double chiBlockSqdSum; // accumulate block <chi>^2 values
int stepInBlock; // number of steps in current block
int blocks; // number of blocks

The following function can be called to initialize the values of the variables.

wolff.cpp

void initializeObservables() {
    chiSum = chiSqdSum = 0;
nChi = 0;
    chiBlock = chiBlockSum = chiBlockSqdSum = 0;
    stepInBlock = blocks = 0;
cChiSum = 0;
cChi = new double [nSave + 1];
    for (int i = 0; i <= nSave; i++)
        cChi[i] = 0;
    nCorr = 0;
}
After each Monte Carlo step, the following function is called to measure the magnetization $M = \sum_i s_i$. If the magnetic field $H = 0$, then the average magnetization $\langle M \rangle = 0$ by symmetry, and the average susceptibility per spin is given by

$$\chi = \frac{1}{N} \langle M^2 \rangle .$$

```cpp
void measureObservables() {
    // observables are derived from the magnetic moment
    int M = 0;
    for (int i = 0; i < Lx; i++)
        for (int j = 0; j < Ly; j++)
            M += s[i][j];
    chi = M * double(M) / double(N);
}
```

The following code accumulates $\chi$ and $\chi^2$ values needed to compute the Monte Carlo error estimate at the end of the run:

```cpp
// accumulate values
chiSum += chi;
chiSqdSum += chi * chi;
++nChi;
```
To measure the auto-correlation time $\tau_\chi$ we need to save $n_{\text{Save}}$ previous values of $\chi$ in the list $\chi_{\text{Save}}$, and accumulate the products $\chi(t)\chi(t-i)$ in the array $c_{\text{Chi}}$. Note the use of an iterator to walk through the list: $\text{iter}$ is essential a pointer to an item saved in the list $\chi_{\text{Save}}$; $*\text{iter}$ fetches the value saved at that item; and using the rules for operator precedence in C/C++, $*\text{iter}++$ parses as $(*(\text{iter}++))$, i.e., increment the pointer after dereferencing its current value.

```cpp
// accumulate correlation values
if (chiSave.size() == nSave) {
    cChiSum += chi;
    cChi[0] += chi * chi;
    ++nCorr;
    list<double>::const_iterator iter = chiSave.begin();
    for (int i = 1; i <= nSave; i++)
        cChi[i] += *iter++ * chi;
    chiSave.pop_back(); // remove oldest saved chi value
}
chiSave.push_front(chi); // add current chi value
```

The errors in a Monte Carlo simulation can be estimated by data-blocking. Suppose that 10,000 configurations are generated by the program. These are divided into 10 blocks of 1,000 configurations each. The average value of $\chi$ is computed in each block, and the Monte Carlo error is estimated as the standard deviation of these average values divided by the square root of the number of blocks. To implement this estimate, we need to
• accumulate $\chi$ values inside each block, and
• compute the block average $\bar{\chi}$, and accumulate $\bar{\chi}$ and $\bar{\chi}^2$ at the end of each block to compute the standard deviation.

```cpp
// accumulate block values
chiBlock += chi;
++stepInBlock;
if (stepInBlock == stepsPerBlock) {
    chiBlock /= stepInBlock;
    chiBlockSum += chiBlock;
    chiBlockSqdSum += chiBlock * chiBlock;
    ++blocks;
    stepInBlock = 0;
    chiBlock = 0;
}
```

**Computing the averages of observables**

At the end of the run, we can use the accumulated measurements to compute various averages:

```cpp
// averages of observables
```
double chiAve; // average susceptibility per spin
double chiError; // Monte Carlo error estimate
double chiStdDev; // Standard deviation error from blocking
double tauChi; // autocorrelation time
double tauEffective; // effective autocorrelation time

void computeAverages() {

    // average susceptibility per spin
    chiAve = chiSum / nChi;

    // Monte Carlo error estimate
    chiError = chiSqdSum / nChi;
    chiError = sqrt(chiError - chiAve * chiAve);
    chiError /= sqrt(double(nChi));

    To measure the auto-correlation time, we use the exponential definition given in Eq. (7.73) of Thijsen's textbook:

    \[
    \tau_{\text{exp}} = -\frac{t}{\log\left|\frac{c_{xx}(t)}{c_{xx}(0)}\right|}.
    \]

    This estimate is averaged over all times for which \( \frac{c_{xx}(t)}{c_{xx}(0)} \) remains larger than a small value which we take to be 0.01. Wolff’s paper uses a more detailed analysis to get a more accurate estimate.
// exponential correlation time
tauChi = 0;
double cAve = cChiSum / nCorr;
double c0 = cChi[0] / nCorr - cAve * cAve;
for (int i = 1; i <= nSave; i++) {
    double c = (cChi[i] / nCorr - cAve * cAve) / c0;
    if (c > 0.01) {
        tauChi += -i/log(c);
    } else {
        tauChi /= (i - 1);
        break;
    }
}
if (i == nSave)
    tauChi /= nSave;

It is straightforward to estimate the standard deviation from the data-blocking:

// standard deviation from blocking
double chiBlockAve = chiBlockSum / blocks;
chiStdDev = chiBlockSqdSum / blocks;
chiStdDev = sqrt(chiStdDev - chiBlockAve * chiBlockAve);
chiStdDev /= sqrt(double(blocks));
Here we compute an effective correlation time defined in Eq. (10) of Wolff’s paper:

$$\tau_{\text{eff}} = \frac{1}{2} \left( \frac{\epsilon_{\text{block}}}{\epsilon_{\text{naive}}} \right)^2,$$

the motivation for which is discussed on page 173 Eq. (7.76) of Thijssen. Basically, if the naive (i.e., Monte Carlo) error estimate does not agree with the data-blocking error estimate, this is an indication that successive configurations are not independent, i.e., the correlation time $\tau > 2$.

```cpp
// effective autocorrelation time
tauEffective = chiStdDev / chiError;
tauEffective *= tauEffective / 2;
```

**The main function**

Finally, the main function steers the simulation.

```cpp
int main() {
    cout << " Two-dimensional Ising Model - Wolff Cluster Algorithm\n";
    cout << " Enter number of spins L in each direction: ";
```
cin >> Lx;
Ly = Lx;
N = Lx * Ly;
cout << " Enter temperature T: ";
cin >> T;
cout << " Enter number of Monte Carlo steps: ";
int MCSteps;
cin >> MCSteps;

initialize();
initializeClusterVariables();

As usual, we start by performing some number of thermalization steps to allow the system to come to thermal equilibrium:

wolff.cpp

int thermSteps = MCSteps / 5;
cout << " Performing " << thermSteps << " thermalization steps ..." << flush;
for (int i = 0; i < thermSteps; i++)
    oneMonteCarloStep();

After the thermalization is done, we need to initialize variables for measuring observables. After each Monte Carlo step, the observables are measured, and at the end of the run the averages are computed.
cout << " done\n Performing production steps ..." << flush;
initializeObservables();
for (int i = 0; i < MCSteps; i++) {
    oneMonteCarloStep();
    measureObservables();
}
cout << " done" << endl;
computeAverages();
cout << " \n Average chi per spin = " << chiAve
     << " \n Monte Carlo error estimate = " << chiError
     << " \n Autocorrelation time tau = " << tauChi
     << " \n Std. Dev. using blocking = " << chiStdDev
     << " \n Effective tau = " << tauEffective << endl;