Critical Slowing Down and Cluster Algorithms

Reducing critical slowing down

The Metropolis Monte Carlo method works very well in simulating the properties of the 2-D Ising model. However, close to the Curie temperature T_c, the simulations suffer from *critical slowing down*, and more efficient algorithms are needed. These *cluster algorithms* are useful in many applications involving graphs and lattices, and they are also very interesting to study.

Critical divergences and spin-spin correlations

At the Curie temperature, some observables like the heat capacity per spin and magnetic susceptibility per spin become divergent (infinite) in the thermodynamic limit of an infinite system. These *critical divergences* are due to *long range correlations* between spins.

Consider two spins, s_0 at the origin of coordinates and s_n at some other lattice site labeled by the index n. The *correlation* between the pair of spins is defined to be

$$\langle s_0 s_n \rangle \ . \tag{1}$$

If the two spins are *uncorrelated* then this average will be zero or very small.

At $T = 0$ the spins are all lined up and so

$$\langle s_0 s_n \rangle = 1 \ . \tag{2}$$

However, this is a somewhat trivial correlation because flipping s_0 will hardly affect s_n if it is not a neighbor of s_0.

Near T_c, the situation is very different: the spins are constantly changing, but not independently; there are large domains (droplets) of parallel spins which persist for long periods of time. Thus, spins far apart from one another are strongly correlated.

At high temperatures, the spins fluctuate rapidly but almost independently of other spins.

To describe these properties of spin correlations, it is conventional to define the *pair correlation function* as

$$g(r) = \langle s_0 s_n \rangle - \langle s_0 \rangle \langle s_n \rangle \ , \tag{3}$$

that is, by subtracting out the average values of the spins considered independently. Here we have defined the *distance* between the two spins

$$r = a\sqrt{n_x^2 + n_y^2} \ , \tag{4}$$

where a is the lattice spacing of the 2-D square lattice. For a translationally invariant system, e.g., with the use of periodic boundary conditions on a finite lattice, the choice of s_0 is not significant: any other spin would do. The important variable is the *distance* r between the two spins. For a large system, r can considered to be a continuous variable.

The pair correlation function can be parametrized as follows for large $r \gg a$:

$$g(r) \sim \frac{e^{-r/\xi}}{r^{d-2+\eta}} \ , \tag{5}$$

where $\xi(T)$ is the *correlation length*, $d = 2$ is the dimensionality of space, and η is a *critical exponent* ($\eta = 1/4$ for the 2-D Ising model).

The correlation length diverges at the critical temperature:

$$\xi(T) \sim \frac{1}{|T - T_c|^{\nu}} , \tag{6}$$

which accounts for long range spin correlations as T approaches T_c. In the 2-D Ising model, $\nu = 1$. At $T = T_c$ the correlation function decays algebraically:

$$g(r) \sim \frac{1}{r^{d-2+\eta}} . \tag{7}$$

Critical slowing down

The Ising model does not have dynamics built into it: there is no kinetic energy term associated with the spins s_i. The Metropolis Monte Carlo method generates successive configurations of spins, but this does not represent the real time evolution of a system of spins.

In a real system, the dynamical variables are functions of time. An interesting quantity is the *relaxation time*, which is the time scale over which the system approaches equilibrium. If $A(t)$ is a quantity which relaxes towards its equilibrium value $\bar{A}$, the the relaxation time can be theoretically as

$$\tau = \frac{\int_0^\infty dt\ t[A(t) - \bar{A}]}{\int_0^\infty dt\ [A(t) - \bar{A}]} . \tag{8}$$

Near the critical temperature, the relaxation time becomes very large and can be shown to diverge for an infinite system:

$$\tau \sim \xi^z \sim \frac{1}{|T - T_c|^{\nu z}} . \tag{9}$$

Here z is the *dynamical critical exponent* associated with the observable A. This phenomenon is called *critical slowing down*.

Autocorrelation time in Metropolis simulations

In a Metropolis simulation, the successive spin configurations also exhibit a type of critical slowing down near the phase transition temperature $T_c(L)$ of the finite lattice. This is not the same as relaxation in a real system. However, it is useful to measure a relaxation time for the Metropolis "dynamics" because it helps to determine how many steps to skip in order to generate *statistically independent* configurations.

Recall that one Monte Carlo step per spin is taken conventionally to be N Metropolis steps. If the correlation time is of the order of a single Monte Carlo step, then every configuration can be used in measuring averages. But if the correlation time is longer, then approximately τ Monte Carlo steps should be discarded between every data point.

The *time autocorrelation function*

$$c_{AA}(k) = \langle (A_n - \langle A_n \rangle)(A_{n+k} - \langle A_{n+k} \rangle) \rangle = \langle A_n A_{n+k} \rangle - \langle A_n \rangle^2 , \tag{10}$$

where n labels the Monte Carlo time step. If Monte Carlo steps separated in time by k intermediate steps are truly uncorrelated, then $c_{AA}(k)$ should be zero (i.e., of $\mathcal{O}(1/\sqrt{M})$ where M is the number of steps used in computing the averages $\langle\rangle$).

If the correlation function decays exponentially

$$c_{AA}(t) \sim e^{-t/\tau} \ , \tag{11}$$

then the *exponential correlation time* can be computed as the average

$$\tau_{\rm exp} = -\left\langle \frac{t}{\log\left|\frac{c_{AA}(t)}{c_{AA}(0)}\right|} \right\rangle \ . \tag{12}$$

If the decay is exponential, then

$$\int_0^\infty dt\ c_{AA}(t) = \int_0^\infty dt\ c_{AA}(0) e^{-t/\tau} = \frac{1}{1/\tau} c_{AA}(0) \ . \tag{13}$$

This suggests another measure of correlation

$$\tau_{\rm int} = \sum_k \frac{c_{AA}(k)}{c_{AA}(0)} \ , \tag{14}$$

which is called the *integrated correlation time*.

In Monte Carlo simulations, the autocorrelation time is often measured as the simulation is running:

- Create an array called `c` with K `double` elements and initialize it to zero.
- Maintain a list of the K most recently computed values of the observable A. This can be an array of length K in which the value of A_n is stored at index `n mod K`.
- At each step $n \geq K$ accumulate the values of $A_{n-K} A_{n-K+k}$ for $k = 0, 1, \ldots K-1$ in the array elements `c[k]`.
- At the end of the run, divide each `c[k]` by $n - K$ and subtract `c[0]`.

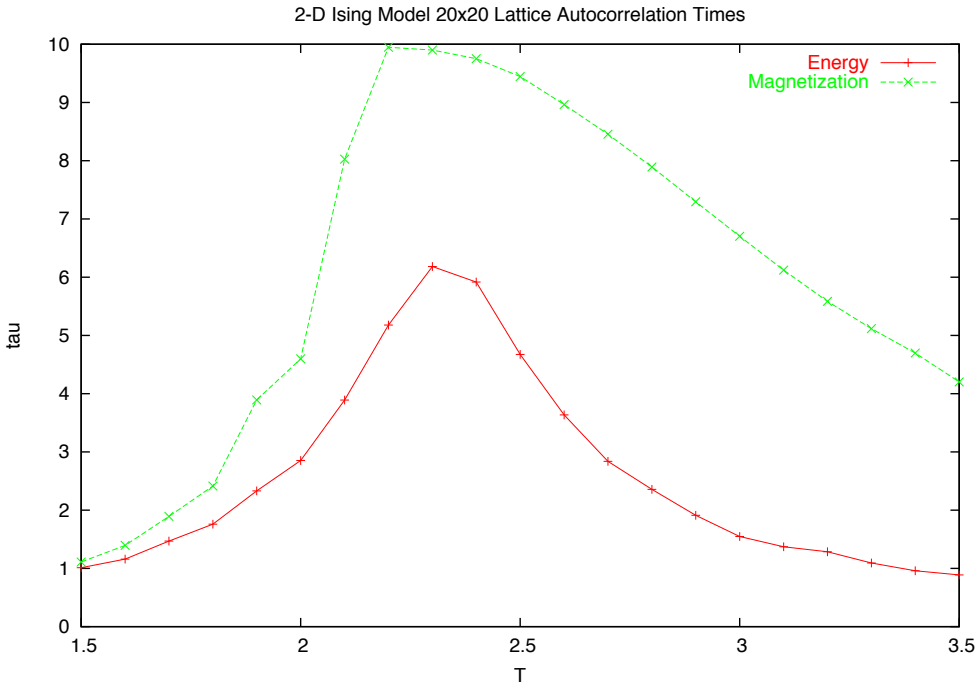
The figure shows measurements of the energy and magnetization autocorrelation times for a 20×20 lattice. Notice that the autocorrelation times become large near the critical temperature $T_{\rm c} = 2.269$ of the infinite system.

Code to measure the autocorrelation time

Program 1: `http://www.physics.buffalo.edu/phy410-505/topic6/auto.cpp`

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <list>                 // to save values for autocorrelations
#include <vector>               // for spin matrix and M & E correlations
using namespace std;

#include "random.hpp"
```




```
using namespace cpl;

Random rg;                          // random number generator
double J = +1;                      // ferromagnetic coupling
int Lx, Ly;                         // number of spins in x and y
int N;                              // number of spins
vector<vector<int> > s;             // the spins
double T;                           // temperature
double H = 0;                       // magnetic field

double w[17][3];                    // Boltzmann factors

void computeBoltzmannFactors ( ) {
     for (int i = -8; i <= 8; i += 4) {
          w[i + 8][0] = exp( - (i * J - 2 * H) / T);
          w[i + 8][2] = exp( - (i * J + 2 * H) / T);
     }
}

double eAv, mAv;                    // accumulators to compute <e> and <m>
int nSave = 10;                     // values to save for autocorrelations
list<double> eSave, mSave;          // saved energy and magnetization values
vector<double> cee, cmm;            // energy and magnetization correlation sums
int nCorr;                          // number of values accumulated in sums

void initializeCorrelations() {
     eAv = mAv = 0;
```

```
     eSave.clear();
     mSave.clear();
     cee = vector<double>(nSave + 1);
     cmm = vector<double>(nSave + 1);
     nCorr = 0;
}

int steps = 0;                  // steps so far
int seed;                       // for random number generator

void initialize ( ) {
     s = vector<vector<int> >(Lx, vector<int>(Ly, 0));
     for (int i = 0; i < Lx; i++)
          for (int j = 0; j < Ly; j++)
               s[i][j] = rg.rand() < 0.5 ? +1 : -1;   // hot start
     computeBoltzmannFactors();
     steps = 0;
}

bool MetropolisStep ( ) {

     // choose a random spin
     int i = int(Lx*rg.rand());
     int j = int(Ly*rg.rand());

     // find its neighbors using 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;

     // find sum of neighbors
     int sumNeighbors = s[iPrev][j] + s[iNext][j] + s[i][jPrev] + s[i][jNext];
     int delta_ss = 2*s[i][j]*sumNeighbors;

     // ratio of Boltzmann factors
     double ratio = w[delta_ss+8][1+s[i][j]];
     if (rg.rand() < ratio) {
          s[i][j] = -s[i][j];
          return true;
     } else return false;
}

double acceptanceRatio;

void oneMonteCarloStepPerSpin ( ) {
     int accepts = 0;
     for (int i = 0; i < N; i++)
          if (MetropolisStep())
               ++accepts;
     acceptanceRatio = accepts/double(N);
```

```
     ++steps;
}

double magnetizationPerSpin ( ) {
     int sSum = 0;
     for (int i = 0; i < Lx; i++)
     for (int j = 0; j < Ly; j++) {
          sSum += s[i][j];
     }
     return sSum / double(N);
}

double energyPerSpin ( ) {
     int sSum = 0, ssSum = 0;
     for (int i = 0; i < Lx; i++)
     for (int j = 0; j < Ly; j++) {
          sSum += s[i][j];
          int iNext = i == Lx-1 ? 0 : i+1;
          int jNext = j == Ly-1 ? 0 : j+1;
          ssSum += s[i][j]*(s[iNext][j] + s[i][jNext]);
     }
     return -(J*ssSum + H*sSum)/N;
}

void accumulateCorrelations() {

     // calculate current energy and magnetization
     double e = energyPerSpin();
     double m = magnetizationPerSpin();

     // accumulate averages and correlation products
     if (eSave.size() == nSave) {   // if nSave values have been saved
          ++nCorr;
          eAv += e;
          mAv += m;
          cee[0] += e * e;
          cmm[0] += m * m;
          list<double>::const_iterator ie = eSave.begin(), im = mSave.begin();
          for (int i = 1; i <= nSave; i++) {
               cee[i] += *ie++ * e;
               cmm[i] += *im++ * m;
          }

          // discard the oldest values
          eSave.pop_back();
          mSave.pop_back();
     }

     // save the current values
     eSave.push_front(e);
     mSave.push_front(m);
```

```
}

double tau_e, tau_m;

void computeAutocorrelationTimes() {

     // energy correlation
     double av = eAv / nCorr;
     double c0 = cee[0] / nCorr - av * av;
     tau_e = 0;
     for (int i = 1; i <= nSave; i++)
          tau_e += (cee[i] / nCorr - av * av) / c0;

     // magnetization correlation
     av = mAv / nCorr;
     c0 = cmm[0] / nCorr - av * av;
     tau_m = 0;
     for (int i = 1; i <= nSave; i++)
          tau_m += (cmm[i] / nCorr - av * av) / c0;
}

int main (int argc, char *argv[]) {

   cout << " Two-dimensional Ising Model - Autocorrelation times\n"
        << " ---------------------------------------------------\n"
        << " Enter number of spins L in each direction: ";
   cin >> Lx;
   Ly = Lx;
   N = Lx * Ly;
   double T1, T2;
   cout << " Enter starting temperature: ";
   cin >> T1;
   cout << " Enter ending temperature: ";
   cin >> T2;
   cout << " Enter number of temperature steps: ";
   int TSteps;
   cin >> TSteps;
   cout << " Enter number of Monte Carlo steps: ";
   int MCSteps;
   cin >> MCSteps;

  // initialize random number generator
  rg.set_seed_time();
  cout << " using " << rg.get_algorithm()
       << " and seed " << rg.get_seed() << endl;

  initialize();
  ofstream dataFile("auto.data");
  int thermSteps = int(0.2 * MCSteps);
  for (int i = 0; i <= TSteps; i++) {
    T = T1 + i * (T2 - T1) / double(TSteps);
```

```
    computeBoltzmannFactors();
    for (int s = 0; s < thermSteps; s++)
      oneMonteCarloStepPerSpin();
      initializeCorrelations();
    for (int s = 0; s < MCSteps; s++) {
      oneMonteCarloStepPerSpin();
      accumulateCorrelations();
    }
    computeAutocorrelationTimes();
    cout << " T = " << T << "\ttau_e = " << tau_e
         << "\ttau_m = " << tau_m << endl;
    dataFile << T << '\t' << tau_e << '\t' << tau_m << '\n';
  }
  dataFile.close();
}
```

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 ξ becomes very large, and the *correlation time* τ, which measures the number of steps between independent Monte Carlo configurations behaves like

$$\tau \sim \xi^z \ , \tag{15}$$

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

The maximum possible value for ξ in a finite system of $N = L \times L$ spins is $\xi \sim L$, because ξ cannot be larger than the lattice size! This implies that $\tau \sim L^{2.1} \simeq N$. 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 ξ 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 $e^{-2J/(k_{\rm B}T)}$, i.e., generate a random deviate r and delete the bond if $r < e^{-2J/(k_{\rm B}T)}$. 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 $1 - e^{-2J/(k_{\rm B}T)}$. 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 ± 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

$$N\tau \sim N\xi^{0.35} \sim NL^{0.35} \sim N^{1.175} \,, \tag{16}$$

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 $\sim 18 \times 18$.

Efficient Cluster Decomposition Algorithms

A lattice with sites and bonds can be viewed as a *graph*, and the problem of finding all clusters is the problem of identifying *connected components* in graph theory.

It is essential to find a cluster labeling algorithm that scales linearly or almost linearly with lattice size N. There are two popular algorithms which have this property:

Backtracking algorithm: This is straightforward to program using a recursive function $f(i)$ where the argument i labels a lattice spin. An array of size N is set up which keeps track of whether a spin s_i has been visited by the backtracking algorithm or not. The algorithm works as follows:

1. 1. Mark all spins as not yet visited.

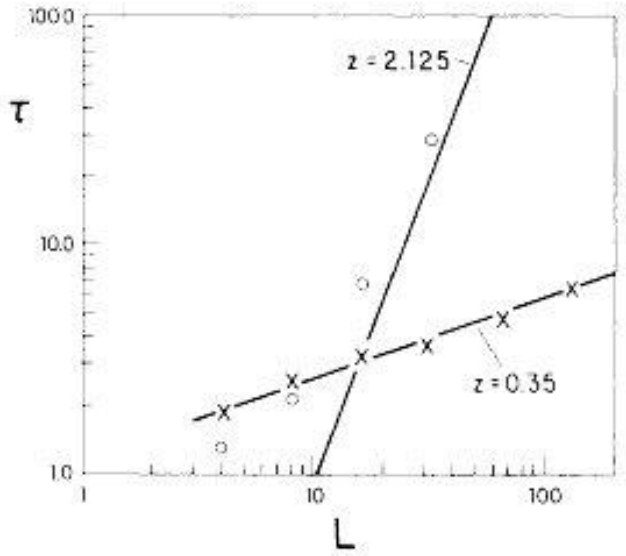


FIG. 1. Log-log plots of correlation times for Monte Carlo simulations of the two-dimensional Ising model at the critical temperature as a function of the linear dimension L. The circles show data for a standard Monte Carlo simulation, and the line marked "$z = 2.125$" gives the expected asymptotic slope (Ref. 4). The crosses show data for the new method, with a least-squares fit labeled with its slope of "$z = 0.35$."

2. 2. Call $f(i)$ on each spin in the lattice. The function does the following:

   - If s_i has been visited, then return. No further testing needed.
   - Otherwise mark s_i as visited.
   - Repeat for each of the 4 nearest neighbors s_j of s_i:
   - If the bond $i-j$ is frozen, then call $f(j)$. This is the recursive step!

A little thought shows that the recursive calls will systematically visit every spin in a cluster connected by frozen bonds. As the lattice is swept, all distinct clusters will be identified.

The recursive function can be given a second argument which can be

an unique cluster number which can be used to mark all spins in the recursive tree, or

the second argument can be a flag to update the cluster spin simultaneously with marking: recall that all spins in a cluster are later updated based on the random test $r < 0.5$.

This backtracking algorithm is easy to understand and program, but is not the most efficient available because many sites will be tested and found to have been visited before.

Hoshen-Koppelman Algorithm: A more efficient algorithm was found by J. Hoshen and R. Kopelman, *Phys. Rev. B* **14**, 3438 (1976). It is somewhat difficult to understand and program. The essential ideas underlying this algorithm are as follows:

The spins (sites) of the lattice are visited in regular order. Thus moving along columns left to right and rows top to bottom, the left and top neighbors of a spin will have been visited, and the right and bottom neighbors will not.

Each spin is assigned a cluster label. If the spin is connected to its left and/or top neighbor by a frozen bond, then it is given the smaller of the two labels. Otherwise, it is given a new unique label larger than all labels assigned so far.

The major problem with this procedure is that clusters will in general be assigned more than one label! To fix this problem, the labels in a cluster are classified as proper or improper (good or bad). A cluster has only one proper label and the others are improper. The algorithm maintains an array A containing "labels of labels" which classify labels into these two categories. This array is indexed by the value of the label. The proper label of a cluster is the minimum value of all of the labels assigned to its spins. When a new label ℓ is created, it is marked as proper by setting $A[\ell] = \ell$. A labeling conflict can arise in scanning the lattice when a spin is connected to *both* its left and top neighbors, and these two neighbors have different labels. When this happens the array value of the larger label is set to the smaller label $A[\ell_{\max}] = \ell_{\min}$. Any label for which $A[\ell] \neq \ell$ is improper. Given an improper label, its proper value can be found using the iteration

While $A[\ell] \neq \ell$ set $\ell = A[\ell]$.

Wolff Single Cluster Algorithm

Two years after Swendsen and Wang published their algorithm, U. Wolff, *Phys. Rev. Lett.* 62, 361 (1989) published an even more efficient algorithm based on constructing and flipping one single cluster at a time.

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

TABLE I. Results for the magnetic susceptibilities χ and autocorrelation times τ_χ (in units comparable to sweeps) for simulations of O(n) σ 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 $\tau_\chi^{\rm eff}$ is directly relevant for error estimation.

n	L	β	$c\times10^{-6}$	$\langle \lvert c \rvert \rangle L^{-2}$	χL^{-2}	τ_χ	$\tau_\chi^{\rm eff}$
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)
2	64	1.07	0.13	0.2629(9)	0.3245(5)		1.1(2)
2	128	1.07	0.10	0.2114(9)	0.2608(5)		1.2(2)
2	128	1.04	0.26	0.1638(5)	0.2032(5)		1.4(2)
				$\langle \lvert c \rvert \rangle$	χ		
3	128	1.5	1.54	132.0(4)	174.4(0.9)		0.25(4)
3	128	1.6	2.00	334.5(7)	444.9(1.4)	1.0(6)	0.40(6)

defines as

$$\chi = \frac{1}{L^2} \left\langle \left[\sum_{i=1}^{L^2} \sigma_i \right]^2 \right\rangle . \tag{17}$$

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

$$\chi = \left(\frac{\partial \langle m \rangle}{\partial H} \right)_T = \left(\frac{\partial \langle \sum_i s_i \rangle}{\partial H} \right)_T = \frac{1}{k_{\rm B}T} \left[\left\langle \frac{1}{N} \left(\sum_{i=1}^{N} s_i \right)^2 \right\rangle - \left(\frac{1}{N} \left\langle \sum_i s_i \right\rangle \right)^2 \right] . \tag{18}$$

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 $k_{\rm B}T$.

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 $1 - e^{-2J/(k_{\rm B}T)}$, i.e., generate a uniform deviate r and if $r < 1 - e^{-2J/(k_{\rm B}T)}$ 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 $r < 1 - e^{-2J/(k_{\rm B}T)}$ 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!

OpenGL Demo Program for Cluster Algorithms

The following OpenGL program implements the Swendsen-Wang and Wolff cluster algorithms for the 2-D Ising model on a 400×400 lattice.

Program 2: `http://www.physics.buffalo.edu/phy410-505/topic6/cluster.cpp`

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>
#include <sstream>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include "Basic.hpp"

int L = 400;
int N = 400 * 400;

int **spin, **cluster, **stack, ***bond;
int *prev, *next, *labelLabel;
int stackp;

double J = 1;
double T = 2.25;
double H = 0;
double Tmin = 0.5, Tmax = 4.0, Tc = 2.267;
double w[17][3], expMinus2JOverT;

void computeBoltzmannFactors();
void initialize();
void oneMetropolisSweep();
void oneSwendsenWangStep();
void oneWolffStep();

int algorithm = 0;
void (*oneMonteCarloStep)() = oneMetropolisSweep;
int seed;

void initialize (void) {
     int i, j, x, y;

     N = L * L;
     spin = new int* [L];
     cluster = new int* [L];
     bond = new int** [L];
     for (i = 0; i < L; i++) {
          spin[i] = new int [L];
          cluster[i] = new int [L];
          bond[i] = new int* [L];
          for (j = 0; j < L; j++)
```

```
                bond[i][j] = new int [2];
        }
        prev = new int [L];
        next = new int [L];
        labelLabel = new int [N];
        stack = new int* [N];
        for (i = 0; i < N; i++)
             stack[i] = new int[2];

        seed = cpl::timeSeed();
        for (x = 0; x < L; x++)
             for (y = 0; y < L; y++)
                  spin[x][y] = cpl::rg.rand() > 0.5 ? +1 : -1;

        for (i = 0; i < L; i++) {
             prev[i] = i - 1;
             next[i] = i + 1;
        }
        prev[0] = L - 1;
        next[L - 1] = 0;

        computeBoltzmannFactors();
}

void computeBoltzmannFactors () {
        int i;

        for (i = -8; i <= 8; i += 4) {
             w[i + 8][0] = std::exp( - (i * J + 2 * H) / T);
             w[i + 8][2] = std::exp( - (i * J - 2 * H) / T);
        }
        expMinus2JOverT = std::exp(- 2 * J / T);
}

GLubyte *spinImage;

void oneMetropolisSweep (void) {
        int i, x, y, sum_of_neighbors, delta_SS;
        double ratio_of_Boltzmann_factors;

        for (x = 0; x < L; x++) {
             for (y = 0; y < L; y++) {
                  sum_of_neighbors = spin[x][next[y]] + spin[x][prev[y]]
                       + spin[next[x]][y] + spin[prev[x]][y];
                  delta_SS = 2 * spin[x][y] * sum_of_neighbors;
                  ratio_of_Boltzmann_factors = w[delta_SS + 8][spin[x][y] + 1];
                  if (cpl::rg.rand() < ratio_of_Boltzmann_factors)
                       spin[x][y] = -spin[x][y];
                  for (i = 0; i < 3; i++)
                  spinImage[3 * (L * x + y) + i] =
                       (GLubyte) (spin[x][y] > 0 ? 0 : 255);
```

```
          }
     }

     glutPostRedisplay();
}

int proper (int i) {
     while (labelLabel[i] != i)
          i = labelLabel[i];
     return i;
}

void oneSwendsenWangStep () {
     int i, j, x, y, label, minLabel, bonds;
     int xBond[4], yBond[4];
     int **newSpin = stack;
     double freezeProb = 1 - expMinus2JOverT;

     /* freeze or delete bonds */
     for (x = 0; x < L; x++)
          for(y = 0; y < L; y++) {
               bond[x][y][0] = bond[x][y][1] = 0;
               if (spin[x][y] == spin[next[x]][y] && cpl::rg.rand() < freezeProb)
                    bond[x][y][0] = 1;
               if (spin[x][y] == spin[x][next[y]] && cpl::rg.rand() < freezeProb)
                    bond[x][y][1] = 1;
          }

     /* Hoshen-Kopelman identification of clusters */
     label = 0;
     for (x = 0; x < L; x++)
          for (y = 0; y < L; y++) {
               /* find sites connected to x,y by frozen bonds */
               bonds = 0;
               if (x > 0 && bond[x - 1][y][0]) {   /* check x-prev neighbor */
                    xBond[bonds] = prev[x];
                    yBond[bonds++] = y;
               }
               if (x == L - 1 && bond[x][y][0]) {  /* check x-next neighbor */
                    xBond[bonds] = 0;
                    yBond[bonds++] = y;
               }
               if (y > 0 && bond[x][y - 1][1]) {   /* check y-prev neighbor */
                    xBond[bonds] = x;
                    yBond[bonds++] = prev[y];
               }
               if (y == L - 1 && bond[x][y][1]) {  /* check y-next neighbor */
                    xBond[bonds] = x;
                    yBond[bonds++] = 0;
               }
```

```
            if (bonds == 0) {    /* start new cluster */
                cluster[x][y] = label;
                labelLabel[label] = label;
                ++label;
            } else {   /* relabel bonded spins with smallest proper label */
                minLabel = label;
                for (i = 0; i < bonds; i++) {
                    j = proper(cluster[xBond[i]][yBond[i]]);
                    if (minLabel > j)
                        minLabel = j;
                }
                cluster[x][y] = minLabel;
                for (i = 0; i < bonds; i++) {
                    j = cluster[xBond[i]][yBond[i]];
                    labelLabel[j] = minLabel;
                    cluster[xBond[i]][yBond[i]] = minLabel;
                }
            }
        }

    /* set cluster spins randomly up or down */
    for (i = x = 0; x < L; x++)
        for (y = 0; y < L; y++, i++) {
            cluster[x][y] = proper(cluster[x][y]);
            newSpin[i][0] = newSpin[i][1] = 0;
        }
    for (x = 0; x < L; x++)
        for (y = 0; y < L; y++) {
            i = cluster[x][y];
            if (!newSpin[i][0]) {
                newSpin[i][1] = cpl::rg.rand() < 0.5 ? +1 : -1;
                newSpin[i][0] = 1;
            }
            spin[x][y] = newSpin[i][1];
            for (i = 0; i < 3; i++)
            spinImage[3 * (L * x + y) + i] =
                (GLubyte) (spin[x][y] > 0 ? 0 : 255);
        }

    glutPostRedisplay();
}

void tryAdd (int x, int y, int cluster_spin) {
    if (!cluster[x][y] && spin[x][y] == cluster_spin)
        if (cpl::rg.rand() < 1 - expMinus2JOverT) {
            spin[x][y] = -spin[x][y]; /* flip spin */
            cluster[x][y] = 1;        /* mark as cluster*/
            ++stackp;                 /* push onto stack */
            stack[stackp][0] = x;
            stack[stackp][1] = y;
        }
```

```
}

void oneWolffStep (void) {
     int i, x, y, cluster_spin;

     /* zero cluster spin markers */
     for (x = 0; x < L; x++)
          for (y = 0; y < L; y++)
               cluster[x][y] = 0;
     /* initialize stack */
     stackp = -1;

     /* choose a random spin */
     x = (int) (L * cpl::rg.rand());
     y = (int) (L * cpl::rg.rand());

     /* grow cluster */
     cluster_spin = spin[x][y];
     spin[x][y] = -spin[x][y];  /* flip the seed spin */
     cluster[x][y] = 1;         /* mark as cluster */
     ++stackp;                  /* push onto stack */
     stack[stackp][0] = x;
     stack[stackp][1] = y;
     while (stackp > -1) {
          x = stack[stackp][0];      /* pop from stack */
          y = stack[stackp][1];
          --stackp;
          /* try add each neighbor to cluster */
          tryAdd(next[x], y, cluster_spin);
          tryAdd(prev[x], y, cluster_spin);
          tryAdd(x, next[y], cluster_spin);
          tryAdd(x, prev[y], cluster_spin);
     }

     for (x = 0; x < L; x++) {
          for (y = 0; y < L; y++) {
               for (i = 0; i < 3; i++)
               spinImage[3 * (L * x + y) + i] =
                    (GLubyte) (spin[x][y] > 0 ? 0 : 255);
          }
     }

     glutPostRedisplay();
}

void init();
void display();
void reshape(int w, int h);
void mouse(int button, int state, int x, int y);

struct Box {    /* pixel coordinates for mouse events */
```

```
     int left;
     int right;
     int top;
     int bottom;
} T_box, spin_box, algorithm_box[3];

int isInside (struct Box *box, int x, int y) {
     return box->left < x && box->right > x
          && box->top < y && box->bottom > y;
}

void init (void) {
     int x, y;

     spinImage = new GLubyte [L * L * 3];

     glClearColor(1.0, 195 / 255.0, 0.0, 0.0);
     glShadeModel(GL_FLAT);
     glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

     initialize();
}

void drawText(const std::string& str, int x, int y) {
     glRasterPos2i(x, y);
     int len = str.find('\0');
     for (int i = 0; i < len; i++)
         glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[i]);
}

void display (void) {
     int i;
     double w, h, dxy;
     std::string name[3] = {"Metropolis", "Swendsen-Wang", "Wolff"};

     dxy = T_box.bottom - T_box.top;
     glClear(GL_COLOR_BUFFER_BIT);
     glColor3ub(192, 192, 192);
     h = dxy * (T - Tmin) / (Tmax - Tmin);
     glRectf(-30.0, 0.0, -6.0, h);
     glColor3ub(0, 0, 0);
     std::ostringstream os;
     os.precision(3);
     os << T << std::ends;
     drawText(os.str(), -33, int(h) - 12);
     os.seekp(0);
     glColor3f(1.0, 0.0, 0.0);
     h = dxy * (Tc - Tmin) / (Tmax - Tmin);
     glRectf(-33.0, h - 2, -3.0, h + 2);
     w = dxy / 3;
     for (i = 0; i < 3; i++) {
```

```
            if (i == algorithm)
                glColor3ub(230, 0, 0);
            else glColor3ub(0, 230, 0);
            glRectf(i * w + 3, -30.0, (i + 1) * w - 3, -6.0);
            glColor3ub(0, 0, 0);
            os << name[i] << std::ends;
            drawText(os.str(), i * int(w) + 10, -20);
            os.seekp(0);
        }
        glRasterPos2i(0, 0);
        glDrawPixels(L, L, GL_RGB, GL_UNSIGNED_BYTE, spinImage);
        glutSwapBuffers();
        glFlush();
}

void reshape (int w, int h) {
        int min_size = 339, widget_size = 30, pad = 3;
        int i, x, y, dx, dy, dxy, widget_area_size;

        widget_area_size = L > 300 ? L : 300;
        dx = dy = pad + widget_size + pad + L + pad;
        if (dx < min_size)
            dx = min_size;
        if (dy < min_size)
            dy = min_size;

        x = w > dx ? (w - dx) / 2 : 0;
        y = h > dy ? (h - dy) / 2 : 0;
        glViewport((GLint) x, (GLint) y, (GLsizei) dx, (GLsizei) dy);

        T_box.left = x + pad;
        T_box.right = T_box.left + widget_size;
        T_box.bottom = h - y - pad - widget_size - pad;
        T_box.top = T_box.bottom - widget_area_size;

        spin_box.left = x + pad + widget_size + pad;
        spin_box.right = spin_box.left + L;
        spin_box.bottom = T_box.bottom;
        spin_box.top = spin_box.bottom - L;

        dxy = widget_area_size / 3;
        for (i = 0; i < 3; i++) {
            algorithm_box[i].left = spin_box.left + i * dxy + pad;
            algorithm_box[i].right = spin_box.left + (i + 1) * dxy - pad;
            algorithm_box[i].top = spin_box.bottom + pad;
            algorithm_box[i].bottom = algorithm_box[i].top + widget_size;
        }

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        x = y = - pad - widget_size - pad;
```

```
     glOrtho(x, x + dx, y, y + dy, -1.0, 1.0);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
}

int running = 1;

void mouse (int button, int state, int x, int y) {
     int i;

     switch (button) {
     case GLUT_LEFT_BUTTON:
          if (state == GLUT_DOWN) {
               if (isInside(&T_box, x, y)) {
                    T = Tmin + (T_box.bottom - y) * (Tmax - Tmin)
                         / (T_box.bottom - T_box.top);
                    computeBoltzmannFactors();
               } else if (isInside(&spin_box, x, y)) {
                    if (running) {
                         running = 0;
                         glutIdleFunc(NULL);
                    } else {
                         running = 1;
                         glutIdleFunc(oneMonteCarloStep);
                    }
               } else {
                    for (i = 0; i < 3; i++) {
                         if (isInside(&algorithm_box[i], x, y))
                              algorithm = i;
                    }
                    switch (algorithm) {
                    case 0:
                         oneMonteCarloStep = oneMetropolisSweep;
                         break;
                    case 1:
                         oneMonteCarloStep = oneSwendsenWangStep;
                         break;
                    case 2:
                         oneMonteCarloStep = oneWolffStep;
                         break;
                    }
                    glutIdleFunc(oneMonteCarloStep);
               }
          }
          glutPostRedisplay();
          break;
     default:
          break;
     }
}
```

```
int main (int argc, char *argv[]) {
     int size = 300;

     glutInit(&argc, argv);
     if (argc > 1) {
          std::istringstream is(argv[1]);
          is >> L;
          std::cout << "Setting L = " << L << std::endl;
     }
     glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
     if (L > size)
          size = L;
     glutInitWindowSize(size + 39, size + 39);
     glutInitWindowPosition(100, 100);
     glutCreateWindow("2-D Ising Model: Cluster Algorithms");
     init();
     glutDisplayFunc(display);
     glutReshapeFunc(reshape);
     glutMouseFunc(mouse);
     glutIdleFunc(oneMonteCarloStep);
     glutMainLoop();
}
```

Homework Problem

Measure the autocorrelation times for the Metropolis, Swendsen-Wang, and Wolff algorithms at three different temperatures, below, above and near T_c. Use the Wolff algorithm to estimate the value of γ/ν by measuring $\chi(T_c)$ as a function of lattice size.

References

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, "Numerical Recipes in C" (Cambridge University Press 1992), §9.2 Secant Method, False Position Method, and ridder's Method, `http://www.nrbook.com/a/bookcpdf/c9-2.pdf`.