

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

$$\langle s_0 s_n \rangle .$$

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

$$\langle s_0 s_n \rangle = 1 .$$

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

$$g(r) = \langle s_0 s_n \rangle - \langle s_0 \rangle \langle s_n \rangle ,$$

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

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

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 be considered to be a continuous variable.

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

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

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:

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

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:

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

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

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

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

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

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*

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

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

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

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

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

If the decay is exponential, then

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

This suggests another measure of correlation

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

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 $k = 0, 1, \dots, K-1$ in the array elements $c[k]$.
- At the end of the run, divide each $c[k]$ by $n - K$ and subtract $c[0]$.

Code to measure the autocorrelation time

```
// Autocorrelation time in the 2-D Ising Model

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <list>           // to save values for autocorrelations
#include "rng.h"

using namespace std;

double J = +1;           // ferromagnetic coupling
int Lx, Ly;              // number of spins in x and y
int N;                   // number of spins
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);
    }
}
```

```
    }  
}
```

Variables and functions to measure autocorrelation times

```
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  
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();  
    if (cee != NULL) delete [] cee;  
    if (cmm != NULL) delete [] cmm;  
    cee = new double [nSave + 1];  
    cmm = new double [nSave + 1];  
    for (int i = 0; i <= nSave; i++)  
        cee[i] = cmm[i] = 0;  
    nCorr = 0;  
}
```

Continue with functions from `ising.cpp`:

```
int steps = 0;                                // steps so far

void initialize ( ) {
    s = new int* [Lx];
    for (int i = 0; i < Lx; i++)
        s[i] = new int [Ly];
    for (int i = 0; i < Lx; i++)
        for (int j = 0; j < Ly; j++)
            s[i][j] = qadran() < 0.5 ?  +1 :  -1;    // hot start
    computeBoltzmannFactors();
    steps = 0;
}

bool MetropolisStep ( ) {

    // choose a random spin
    int i = int(Lx*qadran());
    int j = int(Ly*qadran());

    // 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 (qadran() < 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;
}
```

Accumulating correlation data

```
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);
}
```

Computing autocorrelation times

```
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;
}
```

Steering the calculation: the main function

The main function starts by getting input from the user.

```
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();
ofstream file("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;
    file << T << '\t' << tau_e << '\t' << tau_m << '\n';
}
file.close();
}
```