

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 χ , and the autocorrelation time τ_χ for this observable are measured at the critical temperature $T_c = 2/\log(1 + \sqrt{2}) = 2.2691853\dots$ of the infinite system.

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

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <list>
#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
int steps;               // number of Monte Carlo steps

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
    steps = 0;
}

```

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 `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 < Ly; j++)
            cluster[i][j] = false;

    // choose a random spin and grow a cluster
    int i = int(qadran() * Lx);
    int j = int(qadran() * 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 are 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 (± 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.

```
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);
}
```

```
    if (!cluster[i][jPrev])
        tryAdd(i, jPrev, clusterSpin);
    if (!cluster[i][jNext])
        tryAdd(i, jNext, clusterSpin);
}
```

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

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

```
void tryAdd(int i, int j, int clusterSpin) {
    if (s[i][j] == clusterSpin)
        if (qdran() < 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 χ ,
- 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.

```
// 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.

```
void initializeObservables() {  
    chiSum = chiSqSum = 0;  
    nChi = 0;  
    chiBlock = chiBlockSum = chiBlockSqSum = 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

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

```
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 χ and χ^2 values needed to compute the Monte Carlo error estimate at the end

of the run:

```
// accumulate values
chiSum += chi;
chiSqdSum += chi * chi;
++nChi;
```

To measure the auto-correlation time τ_χ we need to save `nSave` previous values of χ in the list `chiSave`, and accumulate the products $\chi(t)\chi(t-i)$ in the array `cChi`. Note the use of an iterator to walk through the list: `iter` is essential a pointer to an item saved in the list `chiSave`; `*iter` fetches the value saved at that item; and using the rules for operator precedence in C/C++, `*iter++` parses as `*(iter++)`, i.e., increment the pointer *after* dereferencing its current value.

```
// 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* as explained on page 173 in Thijssen's textbook. 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 χ 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 χ 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.

```
// 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:

```
// 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 Thijssen'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 $\tau > 2$.

```

// effective autocorrelation time
tauEffective = chiStdDev / chiError;
tauEffective *= tauEffective / 2;
}

```

The main function

Finally, the `main` function steers the simulation.

```
int main() {  
  
    cout << " Two-dimensional Ising Model - Wolff Cluster Algorithm\n"  
        << " -----\n"  
        << " 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:

```
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;
}
```