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 $T_c = 2/\log(1+\sqrt{2}) = 2.2691853...$ of the infinite system.

wolff.cpp

```
// 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++)                                          24
        for (int j = 0; j < Ly; j++)                                      25
            s[i][j] = qadran() < 0.5 ?  +1 :  -1;   // hot start          26
    steps = 0;                                                            27
}                                                                         28
```

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

$$1 - e^{-2J/(k_{\rm B}T)} .$$

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.

wolff.cpp

```
bool **cluster;                    // cluster[i][j] = true if i,j belongs   30
double addProbability;             // 1 - e^(-2J/kT)                        31

void initializeClusterVariables() {                                       33

    // allocate 2-D array for spin cluster labels                         35
    cluster = new bool* [Lx];                                             36
    for (int i = 0; i < Lx; i++)                                          37
        cluster[i] = new bool [Ly];                                       38

    // compute the probability to add a like spin to the cluster          40
    addProbability = 1 - exp(-2*J/T);                                     41
}                                                                         42
```

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.

wolff.cpp

```
// declare functions to implement Wolff algorithm                    44
void growCluster(int i, int j, int clusterSpin);                     45
void tryAdd(int i, int j, int clusterSpin);                          46

void oneMonteCarloStep() {                                           48

    // no cluster defined so clear the cluster array                 50
    for (int i = 0; i < Lx; i++)                                     51
    for (int j = 0; j < Lx; j++)                                     52
        cluster[i][j] = false;                                       53

    // choose a random spin and grow a cluster                       55
    int i = int(qadran() * Lx);                                      56
    int j = int(qadran() * Ly);                                      57
    growCluster(i, j, s[i][j]);                                      58

    ++steps;                                                         60
}                                                                    61
```

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 (± 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.

wolff.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);
    if (!cluster[i][jPrev])
        tryAdd(i, jPrev, clusterSpin);
```

```
        if (!cluster[i][jNext])                                    84
            tryAdd(i, jNext, clusterSpin);                         85
    }                                                              86
```

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 (± 1) of the seed spin. The candidate spin is added if

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

wolff.cpp

```
    void tryAdd(int i, int j, int clusterSpin) {                   88
        if (s[i][j] == clusterSpin)                                89
            if (qadran() < addProbability)                         90
                growCluster(i, j, clusterSpin);                    91
    }                                                              92
```

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.

wolff.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];                                    121
        for (int i = 0; i <= nSave; i++)                                  122
            cChi[i] = 0;                                                  123
        nCorr = 0;                                                        124
    }                                                                     125
```

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

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

wolff.cpp

```
    void measureObservables() {                                           127

        // observables are derived from the magnetic moment               129
        int M = 0;                                                        130
        for (int i = 0; i < Lx; i++)                                      131
        for (int j = 0; j < Ly; j++)                                      132
            M += s[i][j];                                                 133
        chi = M * double(M) / double(N);                                  134
```

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

wolff.cpp

```
        // accumulate values                                              135
        chiSum += chi;                                                    136
        chiSqdSum += chi * chi;                                           137
        ++nChi;                                                           138
```

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.

wolff.cpp

```
// accumulate correlation values                                    139
if (chiSave.size() == nSave) {                                      140
    cChiSum += chi;                                                 141
    cChi[0] += chi * chi;                                           142
    ++nCorr;                                                        143
    list<double>::const_iterator iter = chiSave.begin();            144
    for (int i = 1; i <= nSave; i++)                                145
        cChi[i] += *iter++ * chi;                                   146
    chiSave.pop_back();     // remove oldest saved chi value        147
}                                                                   148
chiSave.push_front(chi);    // add current chi value                149
```

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.

wolff.cpp

```
// accumulate block values                                          150
chiBlock += chi;                                                    151
++stepInBlock;                                                      152
```

```
        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:

wolff.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 Thijssen's textbook:

$$\tau_{\exp} = -\frac{t}{\log\left|\frac{c_{\chi\chi}(t)}{c_{\chi\chi}(0)}\right|} \, .$$

This estimate is averaged over all times for which $\frac{c_{\chi\chi}(t)}{c_{\chi\chi}(0)}$ 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.

wolff.cpp

```
// 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:

wolff.cpp

```
        // standard deviation from blocking                                    194
        double chiBlockAve = chiBlockSum / blocks;                             195
        chiStdDev = chiBlockSqdSum / blocks;                                   196
        chiStdDev = sqrt(chiStdDev - chiBlockAve * chiBlockAve);               197
        chiStdDev /= sqrt(double(blocks));                                     198
```

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 $\tau > 2$.

wolff.cpp

```
        // effective autocorrelation time                                      199
        tauEffective = chiStdDev / chiError;                                   200
        tauEffective *= tauEffective / 2;                                      201
    }                                                                          202
```

The `main` function

Finally, the `main` function steers the simulation.

wolff.cpp

```
    int main() {                                                               204

        cout << " Two-dimensional Ising Model - Wolff Cluster Algorithm\n"     206
             << " ---------------------------------------------------\n"      207
             << " Enter number of spins L in each direction:  ";               208
```

```
cin >> Lx;                                                    209
Ly = Lx;                                                      210
N = Lx * Ly;                                                  211
cout << " Enter temperature T: ";                             212
cin >> T;                                                     213
cout << " Enter number of Monte Carlo steps:  ";              214
int MCSteps;                                                  215
cin >> MCSteps;                                               216

initialize();                                                 218
initializeClusterVariables();                                 219
```

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;                                 220
cout << " Performing " << thermSteps                          221
     << " thermalization steps ..." << flush;                 222
for (int i = 0; i < thermSteps; i++)                          223
    oneMonteCarloStep();                                      224
```

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.

wolff.cpp

```
cout << " done\n Performing production steps ..." << flush;   226
initializeObservables();                                      227
for (int i = 0; i < MCSteps; i++) {                           228
    oneMonteCarloStep();                                      229
    measureObservables();                                     230
}                                                             231
```

```
    cout << " done" << endl;                                                    232
    computeAverages();                                                          233
    cout << "\n       Average chi per spin = " << chiAve                        234
         << "\n Monte Carlo error estimate = " << chiError                      235
         << "\n   Autocorrelation time tau = " << tauChi                       236
         << "\n   Std.  Dev.  using blocking = " << chiStdDev                  237
         << "\n              Effective tau = " << tauEffective << endl;        238
}                                                                               239
```