

Code for the Swendsen-Wang algorithm

The following code implements the Swendsen-Wang algorithm for the 2-D Ising model.

```
// Swendsen-Wang cluster algorithm for 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
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
```

```

    steps = 0;
}

```

Variables for cluster algorithms

Recall that there are $2N$ bonds where N is the number of spins. We label bonds with the spin label i, j : an i bond connects to spin $i+1, j$, and a j bond to spin $i, j+1$. The `bool` arrays `iBondFrozen` and `jBondFrozen` mark frozen bonds in the lattice. The 2-D array `cluster` will hold the cluster labels of the spins in the lattice.

The most interesting variable is the array `labelLabel` which has N components. Cluster numbers will be assigned to the spins starting with 0 and increasing to a maximum of $N-1$. It will turn out that the spins in each cluster can have several different labels in this range. However, the label sets in distinct clusters do not overlap, i.e., each cluster has its own unique set of labels. The smallest label value in any set is the *proper label* of that cluster. If a label ℓ belongs to a cluster set, the `labelLabel` $[\ell] = \ell'$, which belongs to the cluster set and is $\leq \ell$. Furthermore, `labelLabel` $[\ell] = \ell$ if and only if ℓ is the proper label of the cluster. This array therefore provides a directed lists of labels in each cluster which terminate on the proper cluster label.

```

bool **iBondFrozen, **jBondFrozen; // bond lattice - two bonds per spin
double freezeProbability;           // 1 - e^(-2J/kT)
int **cluster;                      // cluster labels for spins
int *labelLabel;                    // to determine proper labels
bool *sNewChosen;                   // has the new spin value been chosen?
int *sNew;                          // random new spin values in each cluster

void initializeClusterVariables() {

    // allocate 2-D arrays for bonds in x and y directions
    iBondFrozen = new bool* [Lx];
    jBondFrozen = new bool* [Lx];
}

```

```
for (int i = 0; i < Lx; i++) {
    iBondFrozen[i] = new bool [Ly];
    jBondFrozen[i] = new bool [Ly];
}

// compute the bond freezing probability
freezeProbability = 1 - exp(-2*J/T);

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

// allocate arrays of size = number of spins for
labelLabel = new int [N];           // proper label pointers
sNewChosen = new bool [N];          // setting new cluster spin values
sNew = new int [N];                 // new cluster spin values
}
```

One Swendsen-Wang Monte Carlo step

There are three main steps in the Swendsen-Wang algorithm:

- Construct a bond lattice of frozen or melted bonds.
- The frozen bonds partition the spins into clusters or like spins which are identified and labeled using an efficient cluster-labeling algorithm.
- All spins in each cluster are set randomly to ± 1 .

```
// declare functions to implement Swendsen-Wang algorithm
void freezeOrMeltBonds();
int properLabel(int label);
void labelClusters();
void flipClusterSpins();

void oneMonteCarloStep() {

    // first construct a bond lattice with frozen bonds
    freezeOrMeltBonds();

    // use the Hoshen-Kopelman algorithm to identify and label clusters
    labelClusters();

    // re-set cluster spins randomly up or down
    flipClusterSpins();

    ++steps;
}
```

The following function constructs the bond lattice appropriate to the temperature T .

```
void freezeOrMeltBonds() {

    // visit all the spins in the lattice
    for (int i = 0; i < Lx; i++)
        for (int j = 0; j < Ly; j++) {
```

```

// freeze or melt the two bonds connected to this spin
// using a criterion which depends on the Boltzmann factor
iBondFrozen[i][j] = jBondFrozen[i][j] = false;

// bond in the i direction
int iNext = i == Lx-1 ? 0 : i+1;
if (s[i][j] == s[iNext][j] && qadran() < freezeProbability)
    iBondFrozen[i][j] = true;

// bond in the j direction
int jNext = j == Ly-1 ? 0 : j+1;
if (s[i][j] == s[i][jNext] && qadran() < freezeProbability)
    jBondFrozen[i][j] = true;
}
}

```

Implementing the Hoshen-Kopelman cluster-labeling algorithm

The algorithm assigns integer labels to each spin in a cluster. Each cluster has its own distinct set of labels. The following function finds the *proper* label of a cluster, which is defined to be the smallest label of any spin in the cluster. Labels can take integer values $0, 1, 2, \dots, N-1$, where N is the number of spins. The `int` array `labelLabel` has N elements. If `label` is a label belonging to a cluster, the `labelLabel[label]` is the index of another label in the *same* cluster which has a smaller value if such a smaller value exists. Thus, evaluating `labelLabel[label]` repeatedly will find the proper label for the cluster.

```

int properLabel(int label) {
    while (labelLabel[label] != label)
        label = labelLabel[label];
}

```

```
    return label;
}
```

The Hoshen-Kopelman algorithm is implemented in the following function:

```
void labelClusters() {

    int label = 0;

    // visit all lattice sites
    for (int i = 0; i < Lx; i++)
        for (int j = 0; j < Ly; j++) {

            // find previously visited sites connected to i,j by frozen bonds
            int bonds = 0;
            int iBond[4], jBond[4];

            // check bond to i-1,j
            if (i > 0 && iBondFrozen[i - 1][j]) {
                iBond[bonds] = i - 1;
                jBond[bonds++] = j;
            }

            // apply periodic conditions at the boundary:
            // if i,j is the last site, check bond to i+1,j
            if (i == Lx - 1 && iBondFrozen[i][j]) {
                iBond[bonds] = 0;
                jBond[bonds++] = j;
            }
        }
    }
```

```
}

// check bond to i,j-1
if (j > 0 && jBondFrozen[i][j - 1]) {
    iBond[bonds] = i;
    jBond[bonds++] = j - 1;
}

// periodic boundary conditions at the last site
if (j == Ly - 1 && jBondFrozen[i][j]) {
    iBond[bonds] = i;
    jBond[bonds++] = 0;
}

// check number of bonds to previously visited sites
if (bonds == 0) { // need to start a new cluster
    cluster[i][j] = label;
    labelLabel[label] = label;
    ++label;
} else {          // re-label bonded spins with smallest proper label
    int minLabel = label;
    for (int b = 0; b < bonds; b++) {
        int pLabel = properLabel(cluster[iBond[b]][jBond[b]]);
        if (minLabel > pLabel)
            minLabel = pLabel;
    }

    // set current site label to smallest proper label
    cluster[i][j] = minLabel;
}
```

```

        // re-set the proper label links on the previous labels
        for (int b = 0; b < bonds; b++) {
            int pLabel = cluster[iBond[b]][jBond[b]];
            labelLabel[pLabel] = minLabel;

            // re-set label on connected sites
            cluster[iBond[b]][jBond[b]] = minLabel;
        }
    }
}

```

Generating the next system configuration

This is done by setting all spins of each cluster randomly to ± 1 .

```

void flipClusterSpins() {

    for (int i = 0; i < Lx; i++)
        for (int j = 0; j < Ly; j++) {

            // random new cluster spins values have not been set
            int n = i * Lx + j;
            sNewChosen[n] = false;

            // replace all labels by their proper values
            cluster[i][j] = properLabel(cluster[i][j]);
        }
    }
}

```

```
}

int flips = 0;    // to count number of spins that are flipped
for (int i = 0; i < Lx; i++)
for (int j = 0; j < Ly; j++) {

    // find the now proper label of the cluster
    int label = cluster[i][j];

    // choose a random new spin value for cluster
    // only if this has not already been done
    if (!sNewChosen[label]) {
        sNew[label] = qdran() < 0.5 ?  +1 :  -1;
        sNewChosen[label] = true;
    }

    // re-set the spin value and count number of flips
    if (s[i][j] != sNew[label]) {
        s[i][j] = sNew[label];
        ++flips;
    }
}
}
```

Observables

Here we only implement a measurement of the energy per spin and its Monte Carlo error estimate.

```
double eSum;           // accumulator for energy per spin
double eSqdSum;        // accumulator for square of energy per spin
int nSum;              // number of terms in sum

void initializeObservables() {
    eSum = eSqdSum = 0;    // zero energy accumulators
    nSum = 0;             // no terms so far
}

void measureObservables() {
    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]);
        }
    double e = -(J*ssSum + H*sSum)/N;
    eSum += e;
    eSqdSum += e * e;
    ++nSum;
}

double eAve;           // average energy per spin
double eError;         // Monte Carlo error estimate

void computeAverages() {
    eAve = eSum / nSum;
```

```
eError = eSqdSum / nSum;  
eError = sqrt(eError - eAve*eAve);  
eError /= sqrt(double(nSum));  
}
```

The main function

```
int main() {  
  
    cout << " Two-dimensional Ising Model - Swendsen-Wang 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();  
  
    int thermSteps = MCSteps / 5;  
    cout << " Performing " << thermSteps  
          << " thermalization steps ..." << flush;
```

```
for (int i = 0; i < thermSteps; i++)  
    oneMonteCarloStep();  
cout << " done\n Performing production steps ..." << flush;  
  
initializeObservables();  
for (int i = 0; i < MCSteps; i++) {  
    oneMonteCarloStep();  
    measureObservables();  
}  
cout << " done" << endl;  
computeAverages();  
cout << " Energy per spin = " << eAve << " +- " << eError << endl;  
}
```