Code for the Swendsen-Wang algorithm

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

swendsen-wang.cpp

```
// 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 in any set is the *proper label* of that cluster. If a label ℓ belongs to a cluster set, the `labelLabel[`ℓ`]` $= \ell'$, which belongs to the cluster set and is $\leq \ell$. Furthermore, `labelLabel[`ℓ`]` $= \ell$ 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.

swendsen-wang.cpp

```
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++) {                                    42
        iBondFrozen[i] = new bool [Ly];                               43
        jBondFrozen[i] = new bool [Ly];                               44
    }                                                                 45

    // compute the bond freezing probability                          47
    freezeProbability = 1 - exp(-2*J/T);                              48

    // allocate 2-D array for spin cluster labels                     50
    cluster = new int* [Lx];                                          51
    for (int i = 0; i < Lx; i++)                                      52
        cluster[i] = new int [Ly];                                    53

    // allocate arrays of size = number of spins for                  55
    labelLabel = new int [N];        // proper label pointers         56
    sNewChosen = new bool [N];       // setting new cluster spin values 57
    sNew = new int [N];              // new cluster spin values       58
}                                                                     59
```

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.

swendsen-wang.cpp

```
// declare functions to implement Swendsen-Wang algorithm             61
```

```
    void freezeOrMeltBonds();                                                    62
    int properLabel(int label);                                                  63
    void labelClusters();                                                        64
    void flipClusterSpins();                                                     65

    void oneMonteCarloStep() {                                                   67

        // first construct a bond lattice with frozen bonds                      69
        freezeOrMeltBonds();                                                     70

        // use the Hoshen-Kopelman algorithm to identify and label clusters      72
        labelClusters();                                                         73

        // re-set cluster spins randomly up or down                              75
        flipClusterSpins();                                                      76

        ++steps;                                                                 78
    }                                                                            79
```

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

swendsen-wang.cpp

```
    void freezeOrMeltBonds() {                                                   81

        // visit all the spins in the lattice                                    83
        for (int i = 0; i < Lx; i++)                                             84
        for (int j = 0; j < Ly; j++) {                                           85

            // freeze or melt the two bonds connected to this spin               87
            // using a criterion which depends on the Boltzmann factor           88
```

```
            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 $0, 1, 2, \ldots, N-1$, 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.

swendsen-wang.cpp

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

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

swendsen-wang.cpp

```
void labelClusters() {                                                    109

    int label = 0;                                                        111

    // visit all lattice sites                                            113
    for (int i = 0; i < Lx; i++)                                          114
    for (int j = 0; j < Ly; j++) {                                        115

        // find previously visited sites connected to i,j by frozen bonds 117
        int bonds = 0;                                                    118
        int iBond[4], jBond[4];                                           119

        // check bond to i-1,j                                            121
        if (i > 0 && iBondFrozen[i - 1][j]) {                             122
            iBond[bonds] = i - 1;                                         123
            jBond[bonds++] = j;                                           124
        }                                                                 125

        // apply periodic conditions at the boundary:                     127
        // if i,j is the last site, check bond to i+1,j                   128
        if (i == Lx - 1 && iBondFrozen[i][j]) {                           129
            iBond[bonds] = 0;                                             130
            jBond[bonds++] = j;                                           131
        }                                                                 132

        // check bond to i,j-1                                            134
```

```
if (j > 0 && jBondFrozen[i][j - 1]) {                                    135
    iBond[bonds] = i;                                                    136
    jBond[bonds++] = j - 1;                                              137
}                                                                        138

// periodic boundary conditions at the last site                         140
if (j == Ly - 1 && jBondFrozen[i][j]) {                                  141
    iBond[bonds] = i;                                                    142
    jBond[bonds++] = 0;                                                  143
}                                                                        144

// check number of bonds to previously visited sites                     146
if (bonds == 0) { // need to start a new cluster                         147
    cluster[i][j] = label;                                               148
    labelLabel[label] = label;                                           149
    ++label;                                                             150
} else {          // re-label bonded spins with smallest proper label    151
    int minLabel = label;                                                152
    for (int b = 0; b < bonds; b++) {                                    153
        int pLabel = properLabel(cluster[iBond[b]][jBond[b]]);           154
        if (minLabel > pLabel)                                           155
            minLabel = pLabel;                                           156
    }                                                                    157

    // set current site label to smallest proper label                   159
    cluster[i][j] = minLabel;                                            160

    // re-set the proper label links on the previous labels              162
    for (int b = 0; b < bonds; b++) {                                    163
```

```
                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.

swendsen-wang.cpp

```
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] = qadran() < 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.

swendsen-wang.cpp

```
    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

swendsen-wang.cpp

```
int main() {                                                                    243

    cout << " Two-dimensional Ising Model - Swendsen-Wang Algorithm\n"          245
         << " ------------------------------------------------------\n"        246
         << " Enter number of spins L in each direction:  ";                    247
    cin >> Lx;                                                                  248
    Ly = Lx;                                                                    249
    N = Lx * Ly;                                                                250
    cout << " Enter temperature T: ";                                           251
    cin >> T;                                                                   252
    cout << " Enter number of Monte Carlo steps:  ";                            253
    int MCSteps;                                                                254
    cin >> MCSteps;                                                             255

    initialize();                                                               257
    initializeClusterVariables();                                               258

    int thermSteps = MCSteps / 5;                                               260
    cout << " Performing " << thermSteps                                        261
         << " thermalization steps ..." << flush;                               262
    for (int i = 0; i < thermSteps; i++)                                        263
        oneMonteCarloStep();                                                    264
    cout << " done\n Performing production steps ..." << flush;                 265

    initializeObservables();                                                    267
    for (int i = 0; i < MCSteps; i++) {                                         268
```

```
            oneMonteCarloStep();                                                    269
            measureObservables();                                                   270
        }                                                                           271
        cout << " done" << endl;                                                    272
        computeAverages();                                                          273
        cout << " Energy per spin = " << eAve << " +- " << eError << endl;          274
    }                                                                               275
```