

```

/*****
PHY506 Computational Physics II Project
Changxue Deng
Apr. 2003
-----

```

```

Green Function Monte Carlo Simulation of Hubbard model
-----

```

```

*****/

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include "rng.h"

using namespace std;

////////////////////////////////////
// Seed number for random number generator
////////////////////////////////////
int seed;
////////////////////////////////////

////////////////////////////////////
// MC parameters
////////////////////////////////////
// Number of Monte Carlo steps
int nStep;
// Number of therm steps
int thermStep;
// measure or not
int do_measure;
int print_info;
int slice_mod;
int gCount;
double accept;
////////////////////////////////////

////////////////////////////////////
// Simulation variables and parameters
////////////////////////////////////
// Number of sites in x and y dimensions
int Nx, Ny;
// Number of site
int nSite;
// Hubbard parameter: hopping integral
double Hubbard_T;
// Hubbard parameter: on-site Coulomb repulsion
double Hubbard_U;
// chemical potential
double Hubbard_mu;
// Hubbard-Stratonovich field
int **HS_field;
// exp( $\tau$ *K) and exp(- $\tau$ *K)
double **expTauKin;
// Green function for up and down band
double **greenFuncUp, **greenFuncDn;

```

```

// Dummy matrixs
// dummy has special use, do NOT try to use dummy as your temperature matrix
double **dummy;
// The following matrix can be used as local matrixs at any time temporany
double **dummy1_up, **dummy2_up;
double **dummy1_dn, **dummy2_dn;
// precalculated values
double expTauLambda[3];
double exp2TauLambda[3];
// working vectors used in OneMonteCarloStep
double *gu1, *gu2;
double *gd1, *gd2;
////////////////////////////////////

////////////////////////////////////
// Number of time slice
////////////////////////////////////
int nSlice;
// Inverse temperature
double beta;
// time slice separation
double tau;
////////////////////////////////////

////////////////////////////////////
// Accumulators
////////////////////////////////////
// Occupation number of up spin
double nu;
// Occupation number of down spin
double nd;
// Double occupation number n(up)*n(dn)
double nud;
// Energy per particle
double ene;
// Square of local moments
double sqm;
// Number of accepted steps
int nAccept;
////////////////////////////////////

void OneMonteCarloStep();
void InitHS();
void InitExpKin();
void InverseRealMat(double **, double **, int);
void measure(int);
void MatMult(double **, double **, double **);
void MatMult1(double **, double **);
void MatMult2(double **, double **);
void MatSum(double **, double **, double);
void CalcGreenFunc(int);

void Initialize()
{
    ifstream Input("input.data");

    // Seed number
    seed = 19730821;

    // Default parameters

```

```

Nx = 2;                // Num. of lattice sites
Ny = 1;
beta = 2;              // Inverse temperature
nSlice = 4;           // Num. of time slices
Hubbard_T = 1;        // Hopping term
Hubbard_U = 2;        // Hubbard interaction
Hubbard_mu = 1;       // Half filling
nStep = 10000;        // Num. of MC steps
print_info = 0;       // print info. on the screen ?
slice_mod = 10;       // How often to use the exact Green function ?
gCount = 0;           // Counter

// Initialize the accumulators
nu = 0;
nd = 0;
nud = 0;
ene = 0;
sqm = 0;
nAccept = 0;

// Read from external file
Input>>Nx;
Input>>Ny;
Input>>beta;
Input>>nSlice;
Input>>Hubbard_U;
Input>>nStep;
Input>>print_info;
Input>>slice_mod;

//////////////////////
// Generated parameters
//////////////////////
// Chemical potential
Hubbard_mu = Hubbard_U*0.5; // Hald filling
// Number of sites
nSite = Nx*Ny;
// del tau
tau = beta/nSlice;
// Number of thermlizing steps
thermStep = int(0.2*nStep);

// Pre-calculated exponential values
// cos(tau * lambda) = exp(tau*U/2)
double lambda = log(exp(0.5*tau*Hubbard_U) + sqrt(exp(tau*Hubbard_U)-1));
expTauLambda[0] = exp(-lambda);
expTauLambda[1] = exp(+lambda);
expTauLambda[2] = exp(+lambda);
exp2TauLambda[0] = exp(-2*lambda) - 1;
exp2TauLambda[1] = exp(+2*lambda) - 1;
exp2TauLambda[2] = exp(+2*lambda) - 1;

// Allocate memory
HS_field = new int*[nSlice];
for(int l=0; l<nSlice; l++)
    HS_field[l] = new int[nSite];

expTauKin = new double*[nSite]; // exp(-tau*K)
greenFuncUp = new double*[nSite]; // green function for up band
greenFuncDn = new double*[nSite]; // green function for down band

```

```

// Working matrixes
dummy      = new double*[nSite];
dummy1_up  = new double*[nSite];
dummy2_up  = new double*[nSite];
dummy1_dn  = new double*[nSite];
dummy2_dn  = new double*[nSite];

gu1 = new double[nSite];
gu2 = new double[nSite];
gd1 = new double[nSite];
gd2 = new double[nSite];

for(int i=0; i<nSite; i++) {
    expTauKin[i] = new double[nSite];
    greenFuncUp[i] = new double[nSite];
    greenFuncDn[i] = new double[nSite];
    dummy[i] = new double[nSite];
    dummy1_up[i] = new double[nSite];
    dummy2_up[i] = new double[nSite];
    dummy1_dn[i] = new double[nSite];
    dummy2_dn[i] = new double[nSite];
}

// Initialize the Hubbard-Stratonovich field
InitHS();

// Initialize the expTauKin matrix, the kinetic part.
InitExpKin();

// calculate the initial Green function
CalcGreenFunc(0);
}

void InitHS()
{
    for(int l=0; l<nSlice; l++)
        for(int i=0; i<nSite; i++)
            // Assign the HS field +1 or -1 randomly
            HS_field[l][i] = ran2(seed) > 0.5 ? 1 : -1;
}

void MatSum(double **A, double ** B, double t)
{
    int i, j;
    for(i=0; i<nSite; i++)
        for(j=0; j<nSite; j++)
            A[i][j] = A[i][j] + t*B[i][j];
}

void InitExpKin()
{
    int i, j, k;
    double **K, **Kn;

    K = new double*[nSite];
    Kn = new double*[nSite];
    for(i=0; i<nSite; i++) {
        K[i] = new double[nSite];
        Kn[i] = new double[nSite];
    }
}

```

```

}

// First, set the K matrix, Z = K
if(Ny==1) {
  // 1D matrix
  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++) {
      if(j==i+1 || j==i-1) K[i][j] = -Hubbard_T;
      else K[i][j] = 0;
    }
  // 1D Perodic boundary
  K[nSite-1][0] = -Hubbard_T;
  K[0][nSite-1] = -Hubbard_T;
}

if(Ny>1) {
  // 2D square lattice
  int il, jl, ip, im, jp, jm;
  for(i=0; i<Nx; i++) {
    im = i - 1;
    if(im==-1) im = Nx-1;
    ip = i + 1;
    if(ip==Nx) ip = 0;
    for(j=0; j<Ny; j++) {
      jm = j - 1;
      if(jm==-1) jm = Ny - 1;
      jp = j + 1;
      if(jp==Ny) jp = 0;
      for(il=0; il<Nx; il++) {
        for(jl=0; jl<Ny; jl++) {
          if(il==i && (jl==jm || jl==jp))
            K[j*Ny+i][jl*Ny+il] = -Hubbard_T;
          if(jl==j && (il==im || il==ip))
            K[j*Ny+i][jl*Ny+il] = -Hubbard_T;
        }
      }
    }
  }
}

double dt=1;
// Kn = 1
// expTauKin = 1
for(i=0; i<nSite; i++) {
  for(j=0; j<nSite; j++) {
    expTauKin[i][j] = 0;
    Kn[i][j] = 0;
  }
  expTauKin[i][i] = 1;
  Kn[i][i] = 1;
}

// Taylor expansion of exp(-tau*K) and exp(tau*K)
// K is the kinetic matrix
// Kn = K^n
for(i=1; i<10; i++) {
  MatMult1(Kn,K);
  dt *= -tau/i;
  MatSum(expTauKin, Kn, dt);
}

```

```

// We also include the exp(-tau*mu) term in the K matrix
double tmu = exp(tau*(Hubbard_mu-0.5*Hubbard_U));
for(i=0; i<nSite; i++)
  for(j=0; j<nSite; j++)
    expTauKin[i][j] *= tmu;
}

////////////////////////////////////
// B(l) = exp(-tau*K) * exp(-tau*lambda*HS)
////////////////////////////////////
void CalcB(double **B, int l, int s)
{
  int i, j;

  // Note s*HS_field[l][j] = +1 or -1
  // s*HS_field[l][j]+1 = 0 or 2
  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++)
      B[i][j] = expTauKin[i][j]*expTauLambda[-s*HS_field[l][j]+1];
}

////////////////////////////////////
// A = A1*A2
////////////////////////////////////
void MatMult(double **A1, double **A2, double **A)
{
  int i, j, k;
  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++) {
      A[i][j] = 0;
      for(k=0; k<nSite; k++)
        A[i][j] += A1[i][k] * A2[k][j];
    }
}

////////////////////////////////////
// A1 = A1*A2
////////////////////////////////////
void MatMult1(double **A1, double **A2)
{
  int i, j, k;
  // save A1 first
  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++)
      dummy[i][j] = A1[i][j];

  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++) {
      A1[i][j] = 0;
      for(int k=0; k<nSite; k++)
        A1[i][j] += dummy[i][k]*A2[k][j];
    }
}

////////////////////////////////////
// A2 = A1*A2
////////////////////////////////////
void MatMult2(double **A1, double **A2)

```

```

{
  int i, j, k;
  // save A2 first
  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++)
      dummy[i][j] = A2[i][j];

  for(i=0; i<nSite; i++)
    for(j=0; j<nSite; j++) {
      A2[i][j] = 0;
      for(int k=0; k<nSite; k++)
        A2[i][j] += A1[i][k]*dummy[k][j];
    }
}

////////////////////////////////////
// G(l) = 1/(1 + B(l-1)***B(0)B(L-1)***B(1))
////////////////////////////////////
void CalcGreenFunc(int l)
{
  int i, m;

  // B(L-1)***B(1)
  CalcB(dummy2_up, l, +1);
  CalcB(dummy2_dn, l, -1);
  for(m=l+1; m<nSlice; m++) {
    CalcB(dummy1_up, m, +1);
    CalcB(dummy1_dn, m, -1);
    MatMult2(dummy1_up, dummy2_up);
    MatMult2(dummy1_dn, dummy2_dn);
  }

  // Calculate B(l-1)***B(0)
  for(m=0; m<l; m++) {
    CalcB(dummy1_up, m, +1);
    CalcB(dummy1_dn, m, -1);
    MatMult2(dummy1_up, dummy2_up);
    MatMult2(dummy1_dn, dummy2_dn);
  }
  // 1 + B...B
  for(i=0; i<nSite; i++) {
    dummy2_up[i][i] += 1;
    dummy2_dn[i][i] += 1;
  }
  // inverse
  InverseRealMat(dummy2_up, greenFuncUp, nSite);
  InverseRealMat(dummy2_dn, greenFuncDn, nSite);
}

// Find the green function at next time slice
void NextGreenFunc(int l)
{
  int l1 = l + 1;

  // Apply periodic boundary condition
  if(l1 == nSlice) l1 = 0;
  if(gCount == slice_mod || l1==0) {
    //////////////////////////////////////
    // Calculate the exact green function
    // G(l) = 1/(1+B(0)B(1)***B(l-1)B(L-1)***B(1))
  }
}

```

```

////////////////////////////////////
CalcGreenFunc(l1);
gCount = 0;
}
else {
////////////////////////////////////
// use G(l+1) = B(l) * G(l) * B(l)^-1
////////////////////////////////////

// Calculate B(l) for up and down spins
CalcB(dummy1_up, l, +1);
CalcB(dummy1_dn, l, -1);

// G = B(l)*G
MatMult2(dummy1_up, greenFuncUp);
MatMult2(dummy1_dn, greenFuncDn);

// Calculate B^-1 for up and down spins
InverseRealMat(dummy1_up, dummy2_up, nSite);
InverseRealMat(dummy1_dn, dummy2_dn, nSite);

// G = G*B^-1
MatMult1(greenFuncUp, dummy2_up);
MatMult1(greenFuncDn, dummy2_dn);
}

gCount++;
}

double delta_up, delta_dn, dji;
double R_up, R_dn, prob, factor_up, factor_dn;
void OneMonteCarloStep()
{
    int l, i, j, k;

    // Sweep over all the time slices
    for(l=0; l<nSlice; l++) {
        // Sweep over all the sites
        for(i=0; i<nSite; i++) {
            // Calculate the change of exp(-tau*V) due of a trying flip
            delta_up = exp2TauLambda[-HS_field[l][i]+1];
            delta_dn = exp2TauLambda[+HS_field[l][i]+1];

            // Calculate the ratio of determinant for up spin
            // after to before the flip
            R_up = 1 + (1 - greenFuncUp[i][i])*delta_up;
            R_dn = 1 + (1 - greenFuncDn[i][i])*delta_dn;

            prob = fabs(R_up*R_dn);
            prob = 1/(1+prob);

            if(ran2(seed) < prob) {
                nAccept++;

                // Update the flip at site i
                HS_field[l][i] = -HS_field[l][i];

                // Update the green function due to the flip
                for(j=0; j<nSite; j++) {
                    gul[j] = greenFuncUp[i][j];
                }
            }
        }
    }
}

```

```

    gu2[j] = greenFuncUp[j][i];
    gd1[j] = greenFuncDn[i][j];
    gd2[j] = greenFuncDn[j][i];
}
// Save the common factor
factor_up = delta_up/(1 + (1 - greenFuncUp[i][i])*delta_up);
factor_dn = delta_dn/(1 + (1 - greenFuncDn[i][i])*delta_dn);

for(j=0; j<nSite; j++) {
    if(j == i) dji = 1;
    else dji = 0;
    for(k=0; k<nSite; k++) {
        greenFuncUp[j][k] -= factor_up*(dji-gu2[j])*gul[k];
        greenFuncDn[j][k] -= factor_dn*(dji-gd2[j])*gd1[k];
    }
}
}
}

if(do_measure) measure(l);

// Update the green function at next time slice
NextGreenFunc(l);
}
}

void measure(int l)
{
    int n, np, nm, i, j, ip, im, jp, jm;
    double tu, td, tud, e;

    for(int n=0; n<nSite; n++) {
        tu = 1 - greenFuncUp[n][n];
        td = 1 - greenFuncDn[n][n];
        nu += tu;
        nd += td;
        tud = tu*td;
        nud += tud;
        sqm += 1 - 2*tud;

        // Energy
        if(Ny==1) {
            // 1D case
            np = n + 1;
            nm = n - 1;
            if(np==nSite) np = 0;
            if(nm==-1) nm = nSite-1;
            e = Hubbard_T*(greenFuncUp[n][np] + greenFuncUp[n][nm]);
            e += Hubbard_T*(greenFuncDn[n][np] + greenFuncDn[n][nm]);
        }
        else {
            // 2D or 3D case
            j = int((double) n/Ny);
            i = n - j*Ny;
            ip = i + 1;
            im = i - 1;
            jp = j + 1;
            jm = j - 1;
            if(ip==Nx) ip = 0;
            if(jp==Ny) jp = 0;

```

```

    if(im==-1) im = Nx - 1;
    if(jm==-1) jm = Ny - 1;
    // Kinetic energy in x direction
    e = Hubbard_T*(greenFuncUp[n][j*Ny+ip] + greenFuncUp[n][j*Ny+im]);
    e += Hubbard_T*(greenFuncDn[n][j*Ny+ip] + greenFuncDn[n][j*Ny+im]);
    // Kinetic energy in y direction
    e += Hubbard_T*(greenFuncUp[n][jp*Ny+i] + greenFuncUp[n][jm*Ny+i]);
    e += Hubbard_T*(greenFuncDn[n][jp*Ny+i] + greenFuncDn[n][jm*Ny+i]);
}
// On-site potential energy
e += Hubbard_U*tud;
ene += e;
}

}

void average()
{
    double num;

    accept = (double)nAccept/(nSite*nSlice*(nStep+thermStep));
    num = nStep*nSlice*nSite;
    nu /= num;
    nd /= num;
    nud /= num;
    sqm /= num;
    sqm *= 0.75; // Note sqm=(3/4)*(nu-nd)^2
    ene /= num;
}

void print()
{
    cout<<" -----\n";
    cout<<" Results:\n";
    cout<<" Acceptation rate:          "<<accept<<endl;
    cout<<" Average occupation number:      "<<nu+nd<<endl;
    cout<<" Average double occupationn:    "<<nud<<endl;
    cout<<" Average moment:                "<<sqm<<endl;
    cout<<" Average energy:                "<<ene<<endl;
}

int main()
{
    // Initialize all kinds of parameters and allocate
    Initialize();

    cout<<" -----\n";

    // calculate the CPU time for each sweep
    do_measure = 0;
    clock_t t0 = clock();
    OneMonteCarloStep();
    clock_t t1 = clock();
    cout << " CPU time = "<<double(t1-t0)/CLOCKS_PER_SEC<<" sec per sweep\n";

    // thermal steps
    cout<<" Performing thermlization steps.\n";
    cout<<" ..... \n";
    for(int k=0; k<thermStep; k++) {
        OneMonteCarloStep();
    }
}

```

```

    if(print_info) cout<<" Therm steps: "<<k<<endl;
}

cout<<" -----\n";

// measuring steps
// do measuring
do_measure = 1;
cout<<" Performing "<<nStep<<" MC steps.\n";
cout<<" ..... \n";
for(int k=0; k<nStep; k++) {
    OneMonteCarloStep();
    if(print_info) cout<<" MC steps: "<<k<<endl;
}

// Average observables
average();

// Print out results
print();
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Compute the inverse of a real matrix
// From "Numerical Methods for Physics
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void InverseRealMat(double **Atmp, double **Ainv, int dim)
{
    int i, j, k;
    double scale[dim], b[dim][dim];
    int index[dim];
    //double **Atmp;

    //Atmp = new double*[dim];
    //for(i=0; i<dim; i++) Atmp[i] = new double[dim];

    // Matrix b is initialized to the identity matrix
    // and copy A to Atmp
    for(i=0; i<dim; i++)
        for(j=0; j<dim; j++) {
            b[i][j] = 0;
            //Atmp[i][j] = A[i][j];
        }

    for(i=0; i<dim; i++)
        b[i][i] = 1;

    // Set scale factor, scale[i] = max( |A[i][j]| ) for each row
    double scaleMax;
    for(i=0; i<dim; i++) {
        index[i] = i;
        scaleMax = 0;
        for(j=0; j<dim; j++)
            if(fabs(Atmp[i][j]) > scaleMax) scaleMax = fabs(Atmp[i][j]);
        scale[i] = scaleMax;
    }

    // Loop over rows k = 0, ..., dim-2
    /*int signDet = 1;*/

```

```

double ratio, ratioMax, coeff;
int jPivot, indexJ;
for(k=0; k<dim-1; k++) {
    // Select pivot row from max( |A[j][k]/scale[j]| )
    ratioMax = 0;
    jPivot = k;
    for(i=k; i<dim; i++) {
        ratio = fabs(Atmp[index[i]][k])/scale[index[i]];
        if(ratio > ratioMax) {
            jPivot = i;
            ratioMax = ratio;
        }
    }
}
// Perform pivoting using row indexlist
indexJ = index[k];
if(jPivot != k) {
    indexJ = index[jPivot];
    index[jPivot] = index[k];
    index[k] = indexJ;
    /*signDet *= -1;*/
}
// Perform forward elimination
for(i=k+1; i<dim; i++) {
    coeff = Atmp[index[i]][k]/Atmp[indexJ][k];
    for(j=k+1; j<dim; j++)
        Atmp[index[i]][j] -= coeff*Atmp[indexJ][j];
    Atmp[index[i]][k] = coeff;
    for(j=0; j<dim; j++)
        b[index[i]][j] -= Atmp[index[i]][k]*b[indexJ][j];
}
}

// Compute determinant as product of diagonal elements
/*
double determ = signDet; // sign of determinant
for(i=0; i<dim; i++)
    determ *= A[index[i]][i];
*/

// Perform backsubstitution
double sum;
for(k=0; k<dim; k++) {
    Ainv[dim-1][k] = b[index[dim-1]][k]/Atmp[index[dim-1]][dim-1];
    for(i=dim-2; i>=0; i--) {
        sum = b[index[i]][k];
        for(j=i+1; j<dim; j++)
            sum -= Atmp[index[i]][j]*Ainv[j][k];
        Ainv[i][k] = sum/Atmp[index[i]][i];
    }
}
}
}

```