```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define NSIZE 100

int spin[NSIZE][NSIZE];
int cluster[NSIZE][NSIZE];

inline long irand(void);
inline double r250(void);
long lcmrand(long);
void r250init(void);
static long indexf;
static long indexb;
static long buffer[251];

void Metropolis(double, int);
double Energy(void);
double ExactEnergy(double);
double Magnetization(void);
// long int lcm(void);
double realrand(void);
void Wolff(double, int);
void addcluster(int, int, double, int);

int
main()
{
   int i, j, k, nthermal, naverage, nsweep;
   double beta, TotalEnergy, TotalMagnetization;
   double betaini, betafin, betastep;

   nthermal=50;
   naverage=100;
   nsweep=50;
   betaini=0.44;
   betafin=0.45;
   betastep=0.01;

   r250init();

   // Scelgo la direzione iniziale degli spin a caso
   for(i=0; i<NSIZE; i++) {
     for(j=0; j< NSIZE ; j++) {
        if(realrand()>0.5) {
            spin[i][j] = 1;
         }
         else {
           spin[i][j] = -1;
         }
      }
   }

   printf("\nReticolo degli spin %d x %d\n",NSIZE, NSIZE);
   printf("Passaggi di termalizzazione %d\n",nthermal);
   printf("Media su %d stati\n",naverage);
   printf("calcolata su stati dopo %d iterazioni\n",nsweep);

   fflush(stdout);

   // termalizzazione preliminare
   Metropolis(betaini,5*nthermal);
   printf("\n beta  Energia  E esatta  Magnetizzazione\n");
   for(beta=betaini; beta<=betafin; beta += betastep) {

      // termalizzazione
      Wolff(beta,nthermal);

      // azzero i valori medi di energia e magnetizzazione
      TotalEnergy=0;
      TotalMagnetization=0.;
      for(k=1; k<=naverage; k++) {
         // Wolff(beta,nsweep);
```

```
            // Metropolis(beta,nsweep);
            // Calcolo dei valori indipendenti dell'energia e della magnetizzazione
            // Quindi li sommo.
            TotalEnergy += Energy();
            TotalMagnetization += fabs(Magnetization());
        }

        // divido i valori medi per il numero di stati presi
        // in considerazione
        TotalEnergy /= ((double) naverage);
        TotalMagnetization /= ((double) naverage);

        // stampo i risultati
        printf("%7.4lf %7.4lf %7.4lf %7.4lf\n",
          beta, TotalEnergy, ExactEnergy(beta), fabs(TotalMagnetization));
    }
    printf("\n\n");
    return 0;
}

double
Energy()
{
    int isum, i, j;

    // contributo di ogni legame all'energia. Ogni legame
    // viene contato una volta invece di due, per cui il risultato
    // va moltiplicato per due
    isum=0.;
    for(i=0; i<NSIZE; i++) {
        for(j=0; j<NSIZE-1; j++) {
            isum += spin[i][j]*spin[i][j+1];
        }
        isum += spin[i][NSIZE-1]*spin[i][0];
    }
    for(j=0; j<NSIZE; j++) {
        for(i=0; i<NSIZE-1; i++) {
            isum += spin[i][j]*spin[i+1][j];
        }
        isum += spin[NSIZE-1][j]*spin[0][j];
    }
    isum=2*isum;

    // Energia senza campo magnetico
    return -( (double) isum)/( (double) NSIZE*NSIZE );
}

double
Magnetization()
{
    int isum, i, j;

    isum=0;
    for(i=0; i<NSIZE; i++){
        for(j=0; j< NSIZE; j++) {
            isum += spin[i][j];
        }
    }
    return ( (double) isum)/( (double) NSIZE*NSIZE );
}

void
Wolff(double beta, int ntimes)
{
  int i,j,m,s;
  static double Padd;

  // probabilita' di aggiungere un sito al cluster
  Padd=(1.-exp(-2*beta));

   for(m=1; m<=ntimes; m++) {
      for(i=0; i<NSIZE; i++) {
         for(j=0; j<NSIZE; j++) {
            cluster[i][j]=0;
```

```
            }
            }
    // prendo uno spin a caso
    i=floor(realrand()*NSIZE);
    j=floor(realrand()*NSIZE);

    // guardo quanto vale lo spin
    s=spin[i][j];

    // segnalo questo sito come gia' appartenente al cluster
    cluster[i][j]=1;

    // aggiungi elementi al cluster in modo ricorsivo
    addcluster(i,j,Padd,s);

    // cambia il segno di tutti gli spin nel cluster
    for(i=0; i<NSIZE; i++) {
       for(j=0; j<NSIZE; j++)
          if(cluster[i][j]==1) spin[i][j]= -s;
       }
    }
}

void addcluster(int i, int j, double Padd, int s)
{
    int ip1,im1,jp1,jm1;

    // aggiunge al cluster i primi vicini con probabilita' Padd
    // se gia' non appartengono al cluster e se hanno lo stesso spin
    ip1 = i+1;
    if(ip1==NSIZE) ip1=0;
    jp1 = j+1;
    if(jp1==NSIZE) jp1=0;
    im1 = i-1;
    if(i==0) im1=NSIZE;
    jm1 = j-1;
    if(j==0) jm1=NSIZE;

    if(cluster[ip1][j]==0 && spin[ip1][j]==s) {
       if(realrand()< Padd) {
          cluster[ip1][j]=1;
          addcluster(ip1,j,Padd,s);
       }
    }
    if(cluster[im1][j]==0 && spin[im1][j]==s) {
       if(realrand()< Padd) {
          cluster[im1][j]=1;
          addcluster(im1,j,Padd,s);
       }
    }
    if(cluster[i][jp1]==0 && spin[i][jp1]==s) {
       if(realrand()< Padd) {
          cluster[i][jp1]=1;
          addcluster(i,jp1,Padd,s);
       }
    }
    if(cluster[i][jm1]==0 && spin[i][jm1]==s) {
       if(realrand()< Padd) {
          cluster[i][jm1]=1;
          addcluster(i,jm1,Padd,s);
       }
    }
}

void
Metropolis(double beta,int nsweep)
{
    // esegue n iterazioni con l'algoritmo di Metropolis su tutto il reticolo

    int i, j, k, m, im1, ip1, jm1, jp1;
    int DeltaE;
    double newspin,f4,f2;

    f4=exp(-8*beta);    // k=  4
```

```
   f2=exp(-4*beta);    // k=  2

   // loop per NSIZE spin l volte
   for(m=1; m<= nsweep; m++) {
      for(i=0; i< NSIZE; i++) {
         ip1=(i+1)%NSIZE;
         im1=i-1;
         if(i==0) im1=NSIZE-1;

         for(j=0; j< NSIZE; j++) {
            jp1=(j+1)%NSIZE;
            jm1=j-1;
            if(j==0) jm1=NSIZE;
            k=spin[ip1][j]+spin[im1][j]+spin[i][jp1]+spin[i][jm1];
            DeltaE= 2*spin[i][j]*k;
            newspin = -spin[i][j];
            if(DeltaE <= 0) {
               spin[i][j] = newspin;
            }
            else {
               switch(abs(k)) {
                  case 4:
                     if(f4 > realrand()) spin[i][j] = newspin;
                     break;
                  case 2:
                     if(f2 > realrand()) spin[i][j] = newspin;
                     break;
               }
            }
         }
      }
   }
}

/*
long int lcm()
{
   static long int a=16807, m=2147483647, q=127773, r=2836, k;
   static int ncall=0;

   if(ncall==0) {
      ncall=1;
      k=439863;
   }

   k=a*(k%q) - r*(k/q);   // moltiplicazione modulo m
   if(k < 0) k += m;       // se il risultato e' negativo
                           // aggiungo m
   return k;
}
double realrand()
{
   static long int a=16807, m=2147483647, q=127773, r=2836, k;
   static int ncall=0;
   double fac=1./2147483647.;

   if(ncall==0) {
     ncall=1;
     k=198725;
   }

   k=a*(k%q) - r*(k/q);   // moltiplicazione modulo m
   if(k < 0) k += m;       // se il risultato e' negativo
                           // aggiungo m
   return fac*k;
}
*/

double ExactEnergy(double beta)
{
   int j;
   double q1, q2, h, sum, x;
   static double pihalf=1.570796326794897;
```

```
   q1=2*sinh(2*beta)/pow(cosh(2*beta),2);
   q2=2*pow((tanh(2*beta)),2)-1.;
   h=pihalf/1000.;

   sum=0.;
   for(j=0; j<=1000; j++) {
      x=j*h;
      sum += 1.0/sqrt(1.0-pow(q1*sin(x),2));
   }
   sum=sum-0.5*(1.+1./sqrt(1.-q1*q1));
   sum=sum*h;

   return -2./tanh(2*beta)*(1+q2/pihalf*sum);
}



long
lcmrand(long ix)
{
/*
     the minimal standard prng for 31 bit unsigned integer s
     designed with automatic overflow protection
     uses ix as the seed value if it is greater than zero
     otherwise it is ignored
*/

  long  hi, lo, test;
  static long a=16807, m=2147483647, q=127773, r=2836;
  static long x;

  if ( ix > 0 ) x=ix;

  hi = x / q;
  lo = x % q;
  test = a * lo - r * hi;
  if ( test > 0 ) {
    x = test;
  }
  else {
    x = test + m;
  }

  return x;
}

void
r250init()
{

   long iseed;
   long i, k, mask, msb;
   static long ms_bit=1073741824, all_bits=2147483647,
               half_range=536870912, step=7;

   indexf = 1;
   indexb = 104;
   iseed=1;
   k = iseed;
   for( i = 1; i<= 250; i++)
   {
      buffer[i] = lcmrand(k);
      k = -1;
   }
   for(i = 1; i <= 250; i++)
   {
      if ( lcmrand( -1 ) > half_range ) {
         buffer[i] = buffer[i] ^ ms_bit;
      }
   }

   msb = ms_bit;
   mask = all_bits;
```

```
   for(i = 0; i <= 30; i++) {
      k = step * i + 4;
      buffer[k] = buffer[k] & mask;
      buffer[k] = buffer[k] | msb;
      msb = msb / 2;
      mask = mask / 2;
   }

}

inline long
irand()
{
   static int newrand;

   newrand = buffer[indexf] ^ buffer[indexb];
   buffer[indexf] = newrand;

   indexf++;
   if ( indexf > 250 ) indexf = 1;
   indexb++;
   if ( indexb > 250 ) indexb = 1;

   return newrand;
}

double realrand()
{
   return r250();
}

inline double
r250()
{
   return irand()/2147483647.0;
}
```