Homework9

David Mathews

April 9, 2018

1 Metropolis Algorithm

Each section below represents a different section of the code that was either modified or completed.

PARAMS::PARAMS()

```
PARAMS::PARAMS(){
  ifstream pfin;
  pfin.open("param.dat");
  if (pfin.is_open()) {
    pfin >> Nlin;
    pfin >> beta;
    pfin >> Neql;
    pfin >> Nmcs;
    pfin >> Nbin;
    pfin >> SEED;
    pfin >> latt_;
  }
  else
    {cout << "No input file to read ... exiting!"<<endl;exit(1);}
  pfin.close();
  // print out all parameters for record
  cout << "--- Parameters at input for percolation problem ---"<<endl;
  cout <<"Nlin = "<<Nlin<<"; beta = "<<beta<<endl;
  cout <<"# of equillibrium sweeps = "<<Neql<<"; # of Sweeps/bin = "<<Nmcs<<endl;
  cout <<"# of bins = "<<Nbin<<"; SEED = "<<SEED<<endl;
  cout <<"Lattice Type = "<<latt_<<endl;
};//constructor
```

This section of code is the constructor of the PARAMS class. This class constructor opens the parameter file para.dat, reads in the various values found there to the appropriate variables, then closes the file. It then prints the user the parameters that were input so the user can check to make sure things have been properly handled by the code. To debug this section of code, I made sure the values it printed out matched the values in the parameter file.

LATTICE::LATTICE(const PARAMS& p)

```
LATTICE::LATTICE (const PARAMS& p)
{
  if(p.latt_=="sqlatt_PBC")
    {
      Lx=p.Nlin;Ly=p.Nlin;
      Nsite=Lx*Ly;
      //resize neighbor vector
      nrnbrs.resize(Nsite);
      for(int x = 0;x<Lx;x++){
        for(int y = 0;y<Ly;y++){
          //higher x neigbhor
          nrnbrs.at(x+y*Lx).push_back((x+1)%Lx + y*Lx);
          //higher y neighbor
```

```
            nrnbrs.at(x+y*Lx).push_back(x + (y+1)%Ly*Lx);
            //lower y neighbor
            nrnbrs.at(x+y*Lx).push_back(x + (y-1+Ly)%Ly*Lx);
            //lower x neighbor
            nrnbrs.at(x+y*Lx).push_back((x-1+Lx)%Lx + y*Lx);
          }
        }

    }
  else if(p.latt_=="sqlatt_OBC")
    {
      Lx=p.Nlin;Ly=p.Nlin;
      Nsite=Lx*Ly;
      //resize neighbor vector
      nrnbrs.resize(Nsite);
      for(int x = 0;x<Lx;x++){
        for(int y = 0;y<Ly;y++){
          if(x-1>=0)//lower x neighbor
            nrnbrs.at(x+y*Lx).push_back(x-1+y*Lx);
          if(x+1<Lx)//higher x neighbor
            nrnbrs.at(x+y*Lx).push_back(x+1+y*Lx);
          if(y-1>=0)//lower y neighbor
            nrnbrs.at(x+y*Lx).push_back(x+(y-1)*Lx);
          if(y+1<Ly)//higher y neighbor
            nrnbrs.at(x+y*Lx).push_back(x+(y+1)*Lx);
        }
      }

    }
  else
    {cout <<"Dont know your option for lattice in param.dat ... exiting"<<endl;exit(1);}
}
```

This section of the code is exactly the same as it was in the percolation threshold assignment. To verify this section of code was functioning properly, the LATTICE::print() function was used to verify the lattice had the proper structure.

LATTICE::PRINT()

```
void LATTICE::print()
{
  //THIS FUNCTIONS MAY BE CALLED DURING DEBUGGING TO MAKE SURE LATTICE HAS BEEN DEFINED CORRECTL
  cout <<"---printing out properties of lattice ---"<<endl;
  cout<<"size is  "<<Lx<<"x"<<Ly<<endl;
  cout <<"neighbors are"<<endl;
  for (int site=0;site<Nsite;site++)
    {
      cout <<site<<" : ";
      for (int nn=0;nn<nrnbrs.at(site).size();nn++)
        cout<<nrnbrs.at(site).at(nn)<<" ";
      cout <<endl;
    }
}
```

Again this code is exactly as it was in the previous percolation assignment. It simply prints out the lattice to the user in the terminal for checking that the lattice was properly defined.

ISING_CONF::ISING_CONF

```
ISING_CONF::ISING_CONF(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  //first assign random values of either 0 or 1
  dfout.open("data.out");
  spin.resize(p.Nlin*p.Nlin);
  for(int i = 0;i<p.Nlin*p.Nlin;i++){
    spin.at(i)=ran1.randInt(1);
  }
  //CREATE WEIGHT TABLE
  if(p.latt_=="sqlatt_PBC")
    {
      //weight table, first index is sigma old
      //second index is sigma new
      //third index is sum of surrounding indices
      //first resize vector
      wght_tbl.resize(2);
      for(int i = 0;i<2;i++){
        wght_tbl.at(i).resize(2);
        for(int j = 0;j<2;j++){
          wght_tbl.at(i).at(j).resize(5);
        }
      }
      //now iterate through this mess and calculate each value
      for(int i = 0;i<2;i++){
        for(int j = 0;j<2;j++){
          for(int k = 0;k<5;k++){
            wght_tbl.at(i).at(j).at(k)=exp(2.0*p.beta*(j-i)*(2.0*k-4.0));
          }
        }
      }

    }
  else
    {cout <<"NEED_TO_CODE_ALL_LATTICE_OPTIONS"<<endl;}
}
```

This is the constructor for the ISING_CONF class. This first opens the file that will be used for the data. Once "data.out" is opened, then a random spin configuration is created. This spin configuration contains 0's and 1's that will need to be converted during the calculation steps later on. Once this is finished, the weight table is created. This table is a 3 dimensional vector. The first index of this table identifies the old spin, either a 0 or a 1. The second index identifies the new spin, again a 0 or a 1. The third index is for the sum of the surrounding spins. This can be any value between 0 and 4 including 4. At every index in the table, the weight of a spin flip is defined. For any location where the first and second index are the same, this table should store the value 1 as the new configuration is the same as the old configuration in that case. To debug this section of the code, I printed out several different values stored in the weight table and compared them to my own calculations. For the random configuration creation, I simply ran the code several times printing out the initial configuration with the same seed to verify each time it was the same. I then tested with different seeds and saw a different random pattern appear so the function is working properly.

ISING_CONF::conf_write

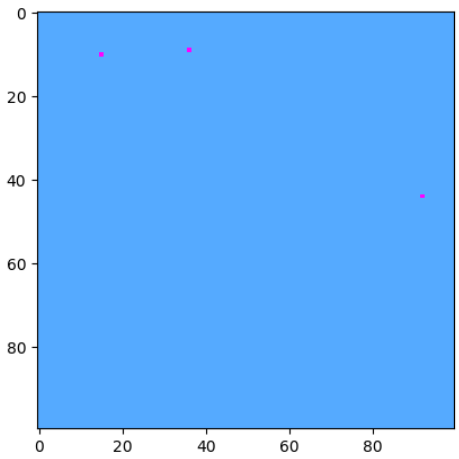
```
void ISING_CONF::conf_write(const PARAMS& p, const LATTICE& latt)
{
  ofstream confout;
  confout.open("blah.out");
  for(int i = 0;i<p.Nlin*p.Nlin;i++){
    if(i%p.Nlin == 0 && i!=0){
      confout<<endl;
    }
    confout<<spin.at(i)<<" ";
  }
  confout<<endl;
  confout.close();
}
```

This function simply writes the configuration to some file in storage. In this version, it writes the current configuration to the file "blah.out". This can easily be changed to any file name desired.

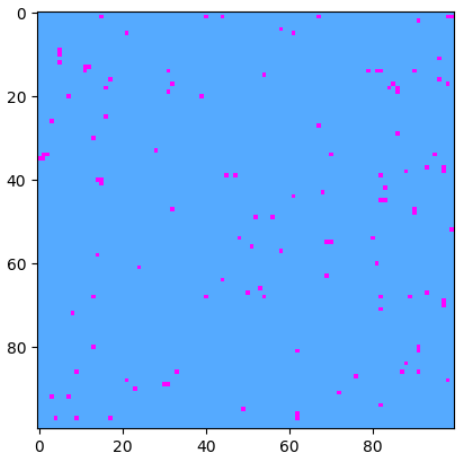
ISING_CONF::sweep

```
void ISING_CONF::sweep(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  /*SWEEP THROUGH THE LATTICE*/
  for(int i = 0;i<p.Nlin*p.Nlin;i++){
    //determine the sum of the neighbors
    int sum = 0;
    for(int j = 0;j<latt.nrnbrs.at(i).size();j++){
      sum+=spin.at(latt.nrnbrs.at(i).at(j));
    }
    double ratio = wght_tbl.at(spin.at(i)).at(-1*spin.at(i)+1).at(sum);
    if(ratio>1.0){//accept this change
      spin.at(i)=-1*spin.at(i)+1;
    }
    else{
      if(ran1.rand()<ratio){
        spin.at(i)=-1*spin.at(i)+1;
      }
    }
  }
}
```

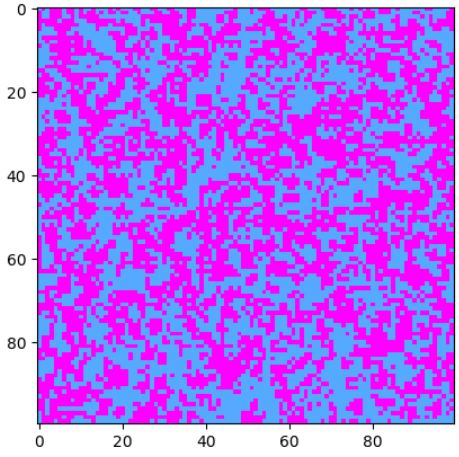
This function determines how the code sweeps across the configuration of spins. In my implementation, the code goes to each index in the spin vector and changes its spin. It then checks to see what the ratio of that change is based on the surrounding spins. If the ratio is greater than 1, the spin flip is accepted and the configuration changes. Otherwise, the spin change is accepted with a probability of R. This check happens once at every index in the spin vector for each sweep. I utilize the equation $-1 * spin + 1$ to convert the spins from $0 \rightarrow 1$ and $1 \rightarrow 0$. To debug this code, I manually calculated a few ratios and compared them to the one my code selected and verified that indeed they matched up. I utilized the ISING_CONF::conf_write function to check the evolution over time. As we have expected behavior for low and high temperatures, I made images of the configuration for low and high temperatures to verify the long term behavior of this sweeping matched what was expected. A few images of this are shown below for a 100x100 configuration.



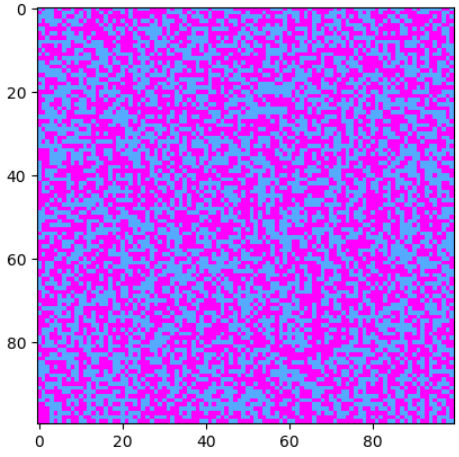
(a) B=1.0



(b) B=0.6



(c) B=0.25



(d) B=0.1

It is clear from these images that the behavior of the sweeping matches the expected behavior. As B increases, the temperature decreases and the image becomes more solid and less random. As B decreases, the temperature increases and the image becomes much more random and scattered.

ISING_CONF::meas_clear

```
void ISING_CONF::meas_clear(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  energy=0.;
  energy_sq=0.;
  mag=0.;
  mag_sq=0.;
}
```

This function resets the values we are measuring. This is called before each new bin of data is calculated.

ISING_CONF::meas

```
void ISING_CONF::meas(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  //remember spin array is 0 and 1, needs to be -1 and 1
  double tempmag=0;
  double tempener=0;
  for(int i=0;i<spin.size();i++){
    //right neighbor energy
    tempener+= (2.0*spin.at(i)-1.0)*(2.0*spin.at(latt.nrnbrs.at(i).at(0))-1.0);
    //upper neighbor energy
    tempener+= (2.0*spin.at(i)-1.0)*(2.0*spin.at(latt.nrnbrs.at(i).at(1))-1.0);
    //magnetization
    tempmag+=2.0*spin.at(i)-1.0;
  }
  double size=spin.size();
  mag+=tempmag/(size);
  energy+=-1.0*tempener/(size);
  mag_sq+=tempmag*tempmag/(size*size);
  energy_sq+=tempener*tempener/(size*size);
}
```

This code is responsible for making the various measurements we want. To do this, I first establish temporary variables to store information in. I then have the code iterate through the spin vector. At each location in the spin vector, the upper neighbor and right neighbor are used to calculate the energy at that location. By using the same two neighbors each time, the whole lattice can be utilized without accidentally repeating a calculation. For the magnetization, the sum of each location in the spin vector is taken. As the spin vector only contains 0's and 1's, the conversion 2*spin-1 is used as that converts the spins to ± 1. Each individual result is added to the appropriate temporary variable. After this loop is completed, I divide these results by the size of the configuration, and then add that result to the total result in the bin.

ISING_CONF::binwrite

```
void ISING_CONF::binwrite(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  mag=mag/(double(p.Nmcs));
  energy=energy/(double(p.Nmcs));
  mag_sq=mag_sq/(double(p.Nmcs));
  energy_sq=energy_sq/(double(p.Nmcs));
  dfout<<energy<<" "<<energy_sq<<" "<<mag<<" "<<mag_sq<<endl;
}
```

This final bit of code prints the results to the file. As each iteration of the ISING_CONF::meas function

Table

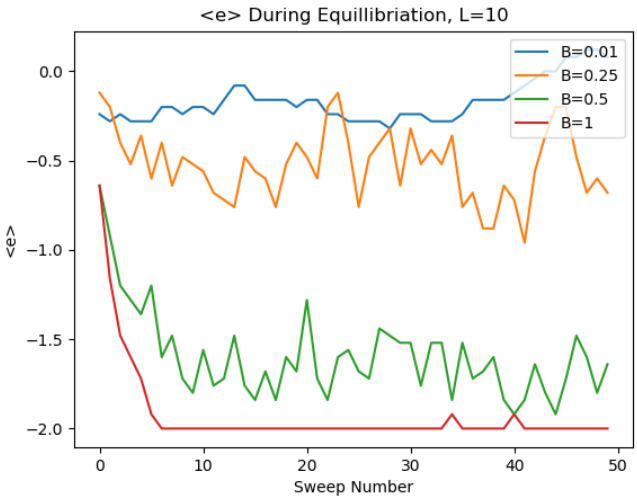
L	β	Neql	Nmcs x Nbin	χ (mc)	χ (enum)	$< e >$ (mc)	e (enum)
4	0.05	10^3	10^7	0.0617860 ± 0.005544	0.06174793	-0.10113 ± 0.0295	-0.10067814
4	0.1	10^3	10^7	0.1566148 ± 0.00935143	0.15661519	-0.205719 ± 0.218068	-0.20571347
4	0.2	10^3	10^7	$0.560974936 \pm 0.024421728$	0.56063833	$-0.456310475 \pm 0.019595$	-0.45613537
3	0.1	10^3	10^6	$0.1557371484 \pm 0.00866583$	0.15583556	-0.22683 ± 0.028659	-0.22526464
3	0.25	10^3	10^6	$0.8794473075 \pm 0.032240937948075$	0.87290363	-0.7542968 ± 0.031664	-0.74700692
16	0.1	10^3	10^6	$0.1565952 \pm 0.010500608$	-	-0.203347 ± 0.007063	-
16	0.25	10^3	10^6	$1.0683776 \pm 0.0521445386368$	-	$-0.557584675 \pm 0.0051801$	-

As hoped, each of the full enumeration results fall within the error bars of the Monte-Carlo method calculations. Interestingly enough, at least in this test, the value for χ and $< e >$ didn't seem to depend very substantially on L. I suspect this changes for very large L values, but at least here it wasn't too substantial.

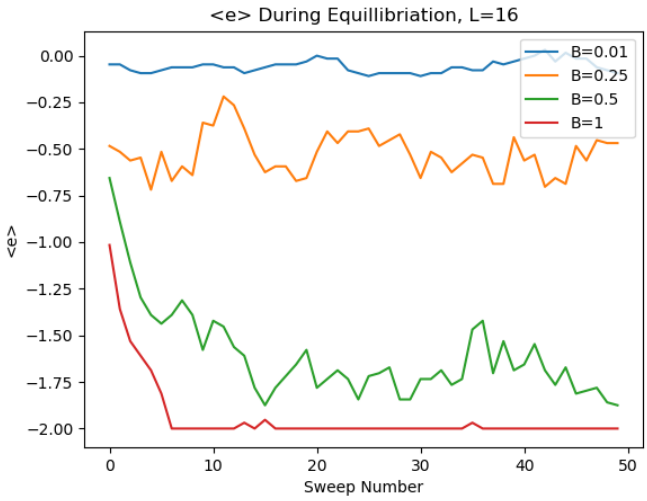
2 MC Histories: Equilibriation, Autocorrelations

2.1 Equilibriation Plots

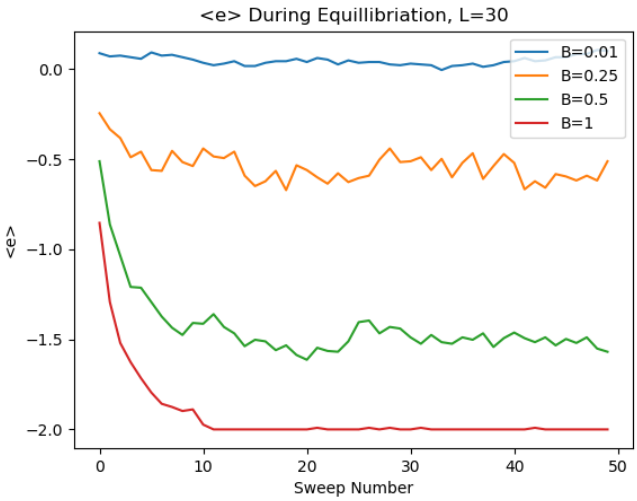
The following plots show the evolution of different spin configurations over the equillibriation process.



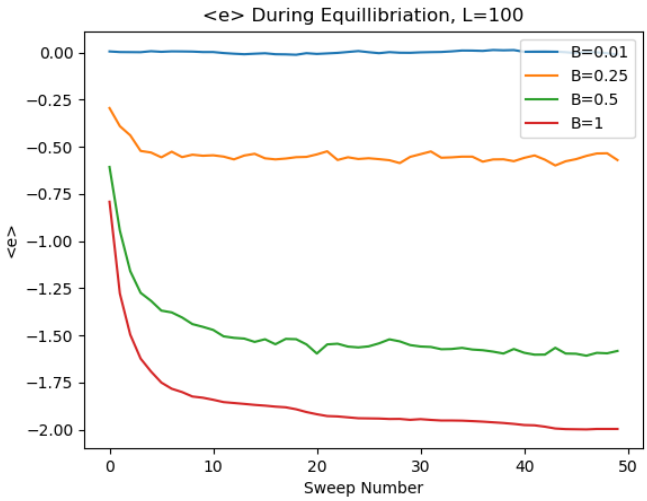
(a) L=10



(b) L=16



(c) L=30



(d) L=100

For these plots, the general trend appears to be that the equillibriation process depends on the size of the lattice. For larger lattices, the overall process took longer. For higher temperatures, or lower B values, the spin configuration is essentially random so it didn't take very long for the configuration to reach equilibrium as we started from a random spin configuration. In the case of large B, or small temperatures, it took substantially longer as the configuration had to "cool" down to the desired temperature and for a lower temperature, this takes longer.

2.2 Equilibriation Movies

The movies themselves are included in the directory I sent with this assignment. To generate these images, I modified the function conf_write functino to be the following.

```
void ISING_CONF::conf_write(const PARAMS& p, const LATTICE& latt, int index)
{
  stringstream ss;
  string file_name;
  ss<<"./movie/conf"<<index<<".spin";
  file_name=ss.str();

  ofstream confout;
  confout.open(file_name.c_str());
  for(int i = 0;i<p.Nlin*p.Nlin;i++){
    if(i%p.Nlin == 0 && i!=0){
      confout<<endl;
    }
    confout<<spin.at(i)<<" ";
  }
  confout<<endl;
  confout.close();
}
```

This required adjusting the definition of the class and several of the function calls within the main function. These changes are in the version of the code attached to this report. Also required was the inclusion of the sstream library to utilize the stringstream function. The movies I attached to this assignment were created with this code. The two movies, called l200b075equil.gif and l200b05equil.gif are gifs of the equillibriation of a 200x200 model with B values of 0.75 and 0.5 respectively. I used these two values as they showed fairly interesting behavior.