Homework10

David Mathews

April 16, 2018

1 Coding

For this assignment, the main code is the same as in Homework 9. The primary changes were made to sweeping function. In this case, the sweep function was changed to being the wolff_update function. This new function is shown below.

```
void ISING_CONF::wolff_update(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  //start cluster from random location
  int startloc=ran1.randInt(latt.Nsite-1);
  //now grow the cluster from this location
  int in=1;
  int out=0;
  //setup the stack
  stack.clear();
  stack.resize(latt.Nsite,out);
  stack.at(0)=startloc;
  int clspin=spin.at(startloc);//spin the whole cluster needs to be
  spin.at(startloc)=(spin.at(startloc)+1)%2;//change start spin now
  stck_end=1;
  stck_pnt=0;
  while(stck_pnt<stck_end){
    int currloc=stack.at(stck_pnt); //current location
    for(int i = 0;i<latt.nrnbrs.at(currloc).size();i++){
      int neigh = latt.nrnbrs.at(currloc).at(i);
      if(clspin==spin.at(neigh)){//if the spin is the same it can join
        if(ran1.rand()<pr){//joins cluster
          stack.at(stck_end)=neigh;
          stck_end++;
          //change spin of this location now to indicate it is in the cluster
          spin.at(neigh)=(spin.at(neigh)+1)%2;
        }
      }
    }
    stck_pnt+=1;
  }
}
```

In the past, we stored the configuration in a vector but in this situation that can be avoided. Nodes of the configuration are either in or out of the cluster, there is no asked and declined option with this algorithm. As such, whenever a node is reached and joins, we can immediately change its spin instead of waiting until the cluster is fully grown. By changing as it goes, the algorithm can avoid asking members of the cluster to join as they are now the opposing spin and therefore can't be re-added. Previously we also had to worry about not asking the same location again, but in this algorithm we only care about asking along the same bond. Due to changing the spin immediately, even if the same bond is considered, it will now have the opposite spin so the issue of considering a bond twice is avoided.

In this algorithm, I utilize several new variables that have been added to the definition of the ISING_CONF class. This new class definition is shown below.

```
class ISING_CONF
{
public:
  ISING_CONF(const PARAMS&, const LATTICE&, MTRand&);//constructor
  void conf_write(const PARAMS&, const LATTICE& latt,int index);
  void wolff_update(const PARAMS&,const LATTICE&,MTRand&);
  void meas_clear(const PARAMS&,const LATTICE&,MTRand&);
  void meas(const PARAMS&,const LATTICE&,MTRand&);
  void binwrite(const PARAMS&,const LATTICE&,MTRand&);
  ~ISING_CONF();// destructor
private:
  vector <int> spin;
  ofstream dfout;
  vector <vector<vector<double> > > wght_tbl;
  // these are the observables
  double energy;
  double energy_sq;
  double mag;
  double mag_sq;
  //this is the setup for the cluster growth
  vector<int> stack;
  int stck_pnt,stck_end;
  double pr;
};
```

Four new variables are added here: stack, stck_pnt, stck_end, and pr. the variable stack stores the sites that have been added to the cluster so that the code can iterate through each one as it progresses. The variables stck_pnt and stck_end store the current location and final location within the stack. Lastly the variable pr stores the probability that a site joins the stack. Both stack and pr are initialized in tbe ISING_CONF constructor as is shown below.

```
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);
  }
  //now initialize the cluster code
  stack.resize(latt.Nsite);
  //joining probability
  pr=1.0-exp(-2.0*p.beta);
}
```

This entire function is exactly the same as it was in Homework 9. The only differences are the addition of the final lines for the cluster code. These lines create a stack that is as large as the whole configuration, which is unnecessarily large, but avoids using append style functions. For pr, its value never changes through the code so it is calculated here once and then accessed later on. As stck_pnt and stck_end are both initialized in the cluster algorithm, those are not included here.

The rest of the code is essentially the same. Any references to the previous sweep function have been removed and replaced with the new wolff_update code but the rest is identical. The full code is attached at the end of this paper.

To verify this code and algorithm works properly, I filled out the same table from the last assignment. This is shown below.

Table

L	β	Neql	Nmcs x Nbin	χ (mc)	χ (enum)	$< e >$ (mc)	e (enum)
4	0.05	10^3	10^7	0.0619 ± 0.00578	0.06174793	-0.101 ± 0.0282	-0.10067814
4	0.1	10^3	10^7	0.157 ± 0.0117	0.15661519	-0.206 ± 0.0261	-0.20571347
4	0.2	10^3	10^7	0.560 ± 0.0244	0.56063833	-0.4563 ± 0.019	-0.45613537
3	0.1	10^3	10^6	0.1558 ± 0.0085	0.15583556	-0.2268 ± 0.0286	-0.22526464
3	0.25	10^3	10^6	0.879 ± 0.0322	0.87290363	-0.7542 ± 0.0316	-0.74700692

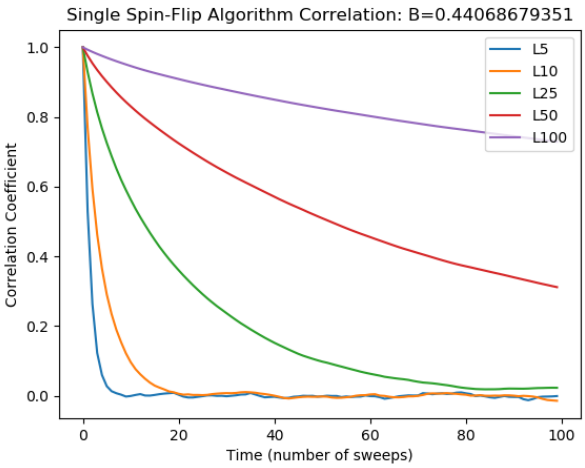
2 Performance Analysis

For this section, the following Python script was written to calculate $A_O(\tau)$.

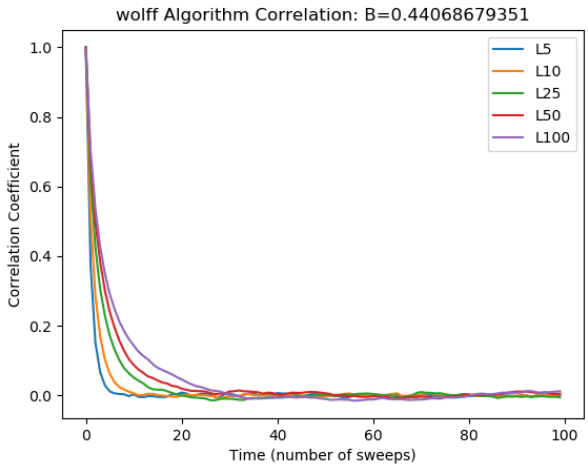
```
#this script will calculate the autocorrelation times
import numpy as np
import matplotlib.pyplot as plt
import os
import sys
import time
from scipy.optimize import curve_fit
code='wolff'
#params
Nlins=[5,10,25,50,100]
beta=np.log(1+np.sqrt(2))/2
Neql=1000
Nmcs=1
Nbin=100000
SEED=100
latt='sqlatt_PBC'
corrtime=100

def func(x,a,b):
  return a*np.exp(-1.0*x/b)

for j in range(len(Nlins)):
  #now call the code to get the data
  #first call the code and get the data values
  print('Running '+str(code) +' with L='+str(Nlins[j]))
  pfile=open('param.dat','w')
  pfile.write('%d %f %d %d %ld %d %s\n' %(Nlins[j],beta,Neql,Nmcs,Nbin,SEED,latt))
  pfile.close()
  os.system('./'+str(code)+' > log.log')
  data=np.loadtxt('data.out')
  print('Analyzing '+str(Nlins[j]))
  #now need to call the code with a huge number of runs
  a=np.zeros(corrtime)
  mags=data[:,3]
  Asquared=np.mean(mags)**2 #average squared
  AA=np.mean(mags*mags)#average of the squared
  corrtimes=np.arange(corrtime)
  for i in range(corrtime):
    #create rolled array
    magst=np.roll(mags,-1*i) #roll the array backwards that many time units
    magst[Nbin-i:]=0
    AB=np.dot(magst,mags)/(float(Nbin-i))#average of the two multiplied together
    a[i]=(AB-Asquared)/(AA-Asquared)
  legend='L'+str(Nlins[j])
  plt.plot(corrtimes,a,label=legend)
  #now do the curve fitting
  res,blah=curve_fit(func,corrtimes,a)
  print('L='+str(Nlins[j])+' : a='+str(res[0])+', b='+str(res[1]))
  #now do integrated time thingy
  print('Theta = '+str(np.sum(a)))
plt.legend(loc='upper right')
plt.title(str(code)+' Algorithm Correlation: B='+str(beta))
plt.xlabel('Time (number of sweeps)')
plt.ylabel('Correlation Coefficient')
```

```
plt.show()
```

This Python script calls the C++ code several times to generate data for different sizes of the model. It then calculates the various averages needed that don't depend on τ. For the value that depends on τ, numpy roll features are used to remove any potential for loops. By simply rolling the array around and setting all of the end values that were wrapped around to 0, for loops can be avoided. Using this code, plots of $A_O(\tau)$ were created for different sizes of spin configurations for both the Single Spin-Flip and the Wolff algorithm.



(a) Single Spin-Flip at the Critical Point



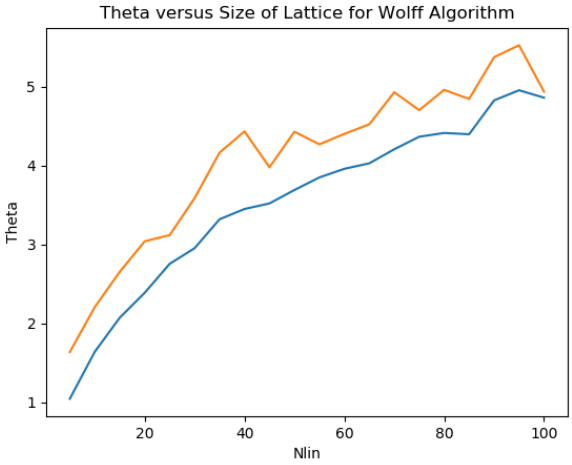
(b) Wolff Algorithm at the Critical Point

It appears based off of these two plots that the two algorithms behave similarly for increasing lattice sizes though the Single Spin-Flip algorithm is substantially worse. To demonstrate this, I used the curve fit routine on each of these results with the relationship $A_O(\tau) = a * e^{-\tau/\theta}$. These results are shown in the table below. For the two different theta values for each test, the Fit Theta came from using the curve fit routine. The Sum Theta values came from using the relationship given in the assignment.

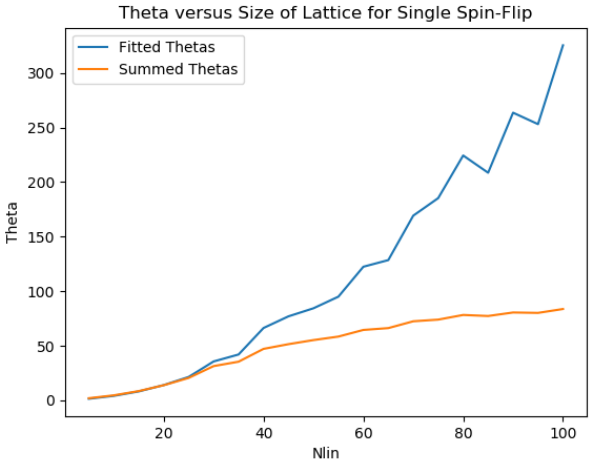
Lattice Size	Wolff Fit Theta	Wolff Sum Theta	Single-Spin Fit Theta	Single-Spin Sum Theta
5	1.04	1.63	1.48	2.0
10	1.63	2.2	4.15	4.65
25	2.75	3.12	21.48	20.57
50	3.688	4.43	84.36	83.71

Clearly from this data the Wolff algorithm performs substantially better around the critical point. The dependence on the lattice size is substantially smaller. Not only that, but the run time for larger lattices is also significantly faster so not only does it take fewer runs to reduce the correlation, those runs go quicker as well.

To determine this dependence, the same Python code was modified to run for substantially more cases of the lattice size and the calculated thetas were plotted and their dependence on lattice size was determined.



(a) For Fit Theta: $z = 0.45$, Sum Theta: $z = 0.35$



(b) For Fit Theta: $z = 1.83$, Sum Theta: $z = 0.87$

It is very clear to see from these plots that the Spin-Flip Method suffers far more from the critical slowing down than the Wolff algorithm. There is an apparent discrepancy in the plot for the Spin-Flip Method due to the innacuracy of the Sum Theta technique. That technique depends on the graph of $A_O(\tau)$ approaching 0 for the max value of τ. In my testing, I did not run all the way to those higher values so this technique became more and more inaccurate for the larger lattices.

3 Full Code

```
#include <fstream>
#include <iostream>
#include <math.h>
#include <string>
#include <iomanip>
#include <vector>
#include <cstdlib>
#include <sstream>
#include "MersenneTwister.h"

using namespace std;

class PARAMS
{
public:
  int Nlin; // linear size of lattice
  double beta; // 1/T
  int Neql;// eql sweeps
  int Nmcs;// sweeps in a bin
  int Nbin;// number of bin
  int SEED; // for MT
  string latt_;// lattice kind
  PARAMS();//constructor
};

class LATTICE
{
public:
  LATTICE(const PARAMS&);//constructor
  int Nsite;// number of lattice sites
  int Lx,Ly;
  vector<vector<int> > nrnbrs;
  void print ();
  string latt_;
};

class ISING_CONF
{
public:
  ISING_CONF(const PARAMS&, const LATTICE&, MTRand&);//constructor
  void conf_write(const PARAMS&, const LATTICE& latt,int index);
  void wolff_update(const PARAMS&,const LATTICE&,MTRand&);
  void meas_clear(const PARAMS&,const LATTICE&,MTRand&);
  void meas(const PARAMS&,const LATTICE&,MTRand&);
  void binwrite(const PARAMS&,const LATTICE&,MTRand&);
  ~ISING_CONF();// destructor
private:
  vector <int> spin;
  ofstream dfout;
  vector <vector<vector<double> > > wght_tbl;
  // these are the observables
  double energy;
  double energy_sq;
  double mag;
  double mag_sq;
```

```
  //this is the setup for the cluster growth
  vector<int> stack;
  int stck_pnt,stck_end;
  double pr;
};

int main(void)
{
  PARAMS p;
  LATTICE latt(p);
  MTRand ran1(p.SEED);
  ISING_CONF ising(p,latt,ran1);
  //ising.conf_write(p,latt);
  //latt.print();
  //EQUILIBRATE
  for(int eql=0;eql<p.Neql;eql++){
    ising.wolff_update(p,latt,ran1);
  }
  //PRODUCTION
  for (int bin=0;bin<p.Nbin;bin++){
    ising.meas_clear(p,latt,ran1);
    for(int mcs=0;mcs<p.Nmcs;mcs++){
      ising.wolff_update(p,latt,ran1);
      ising.meas(p,latt,ran1);
    }
    ising.binwrite(p,latt,ran1);
    ising.conf_write(p,latt,bin);
  }
}

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


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);}
}

void LATTICE::print()
{
  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;
    }
}
```

```
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);
  }
  //now initialize the cluster code
  stack.resize(latt.Nsite);
  //joining probability
  pr=1.0-exp(-2.0*p.beta);
}

ISING_CONF::~ISING_CONF()
{
  dfout.close();
}

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();
}


void ISING_CONF::wolff_update(const PARAMS& p, const LATTICE& latt, MTRand& ran1)
{
  //start cluster from random location
  int startloc=ran1.randInt(latt.Nsite-1);
  //now grow the cluster from this location
  int in=1;
  int out=0;
  //now basically do the same cluster grow algorithm from before
  //setup the stack
  stack.clear();
  stack.resize(latt.Nsite,out);
  stack.at(0)=startloc;
  int clspin=spin.at(startloc);//spin the whole cluster needs to be
  spin.at(startloc)=(spin.at(startloc)+1)%2;//change start spin now
  stck_end=1;
  stck_pnt=0;
  while(stck_pnt<stck_end){
```

```
    int currloc=stack.at(stck_pnt); //current location
    for(int i = 0;i<latt.nrnbrs.at(currloc).size();i++){
      int neigh = latt.nrnbrs.at(currloc).at(i);
      if(clspin==spin.at(neigh)){//spins are the same, it can join
        if(ran1.rand()<pr){//joins cluster
          stack.at(stck_end)=neigh;
          stck_end++;
          //change spin of this location now to indicate it is in the cluster
          spin.at(neigh)=(spin.at(neigh)+1)%2;
        }
      }
    }
    stck_pnt+=1;
  }
}

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

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);
}

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;
}
```