Using Monte Carlo Renormalization Group (MCRG) to study the Ising model on a two dimensional square lattice

Jixun Ding (SUID: 3624401)

March 11, 2019

1 Introduction

Monte Carlo Renormalization Group (MCRG) is a computational method for investigating critical phenomena in thermodynamical systems by combining standard Monte Carlo simulations with real space renormalization group analysis. In this work, I follow a description of MCRG by Swendsen [1] and Binder [2]. I apply MCRG to extract critical exponents of the Ising Model on a two dimensional square lattice. Exact critical exponents for the Ising Model in two dimensions are known [3], so this is a good check for the validity of the MCRG method.

Consider a Hamiltonian of the form

$$\mathcal{H} = \sum_{\alpha} K_\alpha S_\alpha \tag{1}$$

where $\{S_\alpha\}$ are sums of products of spin operators and the $\{K_\alpha\}$ are the corresponding dimensionless coupling constants with factors of $-\beta$ absorbed. Some examples of S_α are:

$$S_h = \sum_i \sigma_i \quad S_{nn} = \sum_{\langle ij \rangle} \sigma_i \sigma_j \quad S_{plaq} = \sum_{\substack{i,j,k,l \\ \text{on a plaquette}}} \sigma_i \sigma_j \sigma_k \sigma_l$$

Applying a renormalization group transformation to the above system integrates out some short range degrees of freedom and produces a new, effective Hamiltonian $\mathcal{H}'$, parametrized by a new set of coupling constants $\{K'_\alpha\}$

MORE THEORY : MC MORE THEORY: RG

$$\sum_{i=0}^{\infty} a_i x^i \tag{2}$$

2 Simulation Method

Consider a system Ω of Ising spins $\sigma_i = \pm 1$ situated on a two dimensional ($d = 2$) square lattice with linear dimension L and lattice spacing 1. Then the total number of lattice sites is $N_s = L^2$. For simplicity, we start our analysis with a microscopic Hamiltonian with only nearest neighbor interactions:

$$\mathcal{H}^{(\prime)} = -\beta H_\Omega = K \sum_{\langle i,j \rangle} \sigma_i \sigma_j + h \sum_{i=0}^{N_s} \sigma_i \tag{3}$$

and set coupling constants to the known critical values ($K = K_c, h = 0$). We use priodic boundary conditions. As discussed previously, because we have started on the critical manifold, as we repeatedly apply renormalization group transformations, the system will be moved towards the critical fixed point $\mathbf{K}^*$. After

applying n RG transformations, we produce the linearized RG matrix $T^{(n)}_{\alpha\beta}$ using Eq. 2, and find its eigenvalues in order to get estimates of critical exponents. As n increases, we expect $T^{(n)}_{\alpha\beta}$ to approach its theoretical value $T_{\alpha\beta}|_{\mathbf{K}^*}$. (With finite precision numbers, we are not *exactly* starting *on* the critical manifold, but we still expect the system to move towards the fixed point $\mathbf{K}^*$ in the few RG iterations that we apply before diverging along one of the relevant directions.)

For the standard Monte Carlo part, I choose to use Wolff's Algorithm [4] to generate spin configurations. [OPTIONAL The average size of spin clusters $\langle c\rangle/N_s$ flipped at each step in Wolff's algorithm is monitored, and] measurements of spin-spin correlation functions are taken every Δ_N steps to ensure that successive measurements are uncorrelated. Δ_N is chosen based on $\langle c\rangle/N_s$. The Monte Carlo simulation settings for a few different lattice sizes are listed in Table 1. A variety of lattice sizes are used because we want to examine how finite lattice sizes impact the renormalization group analysis.

Lattice linear dimension L	64	32	16	8
# of burn-in steps N_{warm}	2×10^4	1×10^4	0.5×10^4	0.5×10^4
# of measurement steps N_{meas}	50×10^4	40×10^4	20×10^4	20×10^4
# of MC steps between measurements Δ_N	10	8	5	5
# of samples $N_{data}=N_{meas}/\Delta_N$	5×10^4	5×10^4	4×10^4	4×10^4

Table 1: Monte Carlo simulation settings

For the renormalization group analysis part, I use a simple block-spin transformation with scale factor $b=2$. The renormalized block-spin value is determined by majority rule, with ties broken by random assignments of $+1$ and -1.

Due to the $\{\sigma_i\}\leftrightarrow\{-\sigma_i\}$ symmetry of our model, we can analyze the even and odd coupling constants in the Hamiltonian separately. In other words, we can suppose that the block spin transformations do not mix even and odd coupling constant spaces. The largest (in magnitude) eigenvalue λ_e of the linearized RG transformation matrix $T_{\alpha\beta}$ for even coupling constants produce the thermal exponent y_T via:

$$y_T=\frac{\log\lambda_e}{\log b} \tag{4}$$

The largest (in magnitude) eigenvalue λ_o of the linearized RG transformation matrix $T_{\alpha\beta}$ for odd coupling constants produce the megnetization exponent y_H via:

$$y_H=\frac{\log\lambda_o}{\log b} \tag{5}$$

From Onsager's exact solution [3] we know the exact critical exponents of the Ising model in two dimensions are $\nu=1,\ \eta=1/4$. So we expect to find

$$y_T=1/\nu=1 \qquad y_H=d-\frac{d-2+\eta}{2}=\frac{15}{8}$$

In order to examine the effect of coupling constant space truncation in evaluating $T_{\alpha\beta}$, coupling constants are added into the analysis one by one. [Equivalently, the $T_{\alpha\beta}$ matrix is of size N_c by N_c, where N_c is the number of coupling constants included in the *RG* analysis.] The even coupling constants that are one-by-one added to the RG analysis are given in Table 2.

Even couplings	
Name	Meaning
K_1	nearest neighbor $(0,0)$ - $(1,0)$
K_2	next-nearest neighbor $(0,0)$ - $(1,1)$
K_3	four spins on a plaquette $(1,0)$ - $(1,1)$ - $(0,1)$ - $(0,0)$
K_4	third nearest neighbor $(0,0)$ - $(2,0)$
K_5	fourth nearest neighbor $(0,0)$ - $(2,1)$
K_6	four spins on a sublattice plaquette $(2,0)$ - $(0,2)$ - $(-2,0)$ - $(0,-2)$
K_7	fifth nearest neighbor $(0,0)$ - $(2,2)$

Table 2: First few even short range coupling constants that may be used in the RG analysis to find y_T

The odd coupling constants that are one-by-one added to the RG analysis are given in Table 3.

Odd couplings	
Name	Meaning
K_1	Magnetization $(0,0)$
K_2	Three spins on a plaquette $(0,0)$ - $(1,0)$ - $(1,1)$
K_3	Three spins in a row $(0,0)$ - $(1,0)$ - $(2,0)$
K_4	Three spins at an angle $(0,0)$ - $(1,0)$ - $(2,1)$

Table 3: First few odd short range coupling constants that may be used in the RG analysis to find y_H

In summary, we are interested in how the size of lattice L, the number of RG iterations n, and the number of coupling constants N_c used in calculating $T_{\alpha\beta}$ affect the y_T, y_H results we obtain from the MCRG method.

3 Results

All results for y_T are shown in Table 4 below.

		Lattice size L			
n	N_c	64	32	16	8
1	1		0.90788777	0.89635236	0.886(2)
1	2		0.9656497	0.96172657	0.962(2)
1	3		0.96821987	0.96560534	0.964(3)
1	4		0.96974414	0.96185282	0.961(3)
1	5		0.96872342	0.96075818	0.954(7)
1	6		0.96883716	0.96134885	0.954(7)
1	7		0.96684275	0.96791929	0.949(6)
2	1				0.88213852
2	2				0.95997822
2	3				0.96333859
2	4				0.95932521
2	5				0.96106577
2	6				0.96105902
2	7				0.96185279
3	1				0.88213852
3	2				0.95997822
3	3				0.96333859
3	4				0.95932521
3	5				0.96106577
3	6				0.96105902
3	7				0.96185279

Table 4: thermal eigenvalue exponent y_T as a function of the number of RG iterations N_r, the number of coupling constants in the RG analysis N_c

From the above, we can see that:
All results for y_H are shown in Table 5 below.

		Lattice size L			
n	N_c	64	32	16	8
1	1		1.88090467	1.8793387	1.87620229
1	2		1.88051803	1.87969459	1.8788031
1	3		1.88052089	1.87973553	1.87884904
1	4		1.88090765	1.88016303	1.87917633
2	1				
2	2				0.95997822
2	3				0.96333859
2	4				0.95932521
3	1				0.88213852
3	2				0.95997822
3	3				0.96333859
3	4				0.95932521

Table 5: thermal eigenvalue exponent y_T as a function of the number of RG iterations N_r, the number of coupling constants in the RG analysis N_c

From the above, we can see that:

References

[1] R. H. Swendsen, *Monte Carlo Renormalization*, pp. 57–84. Topics in current physics, Springer-Verlag, 1982.

[2] D. P. Landau and K. Binder, *Monte Carlo renormalization group methods*, p. 364–377. Cambridge University Press, 4 ed., 2014.

[3] L. Onsager, "Crystal statistics. i. a two-dimensional model with an order-disorder transition," *Phys. Rev.*, vol. 65, pp. 117–149, Feb 1944.

[4] U. Wolff, "Collective monte carlo updating for spin systems," *Phys. Rev. Lett.*, vol. 62, pp. 361–364, Jan 1989.

Appendix: My Code

```
#!/usr/bin/env python
# coding: utf-8

## MCRG Code following Swendsen Description circa 1982
#   Using Wolff's Algorithm to combat critical slowing down

from __future__ import division #safeguard against evil floor division
import numpy as np
from scipy import linalg as la
import matplotlib.pyplot as plt


### Block spin transform, scale factor = b

def assignBlockSpin(total):
    '''Rule for assigning block spin value. Random tiebreaker'''
    if total > 0:
        s = 1;
    elif total < 0:
        s = -1;
    else:
        s = np.random.choice([-1,1])
    return s


def RGTransform(S,b):
    '''Take a spin config S and produce renormalized block spin config that
    groups b*b spins together into one block. Does not modify input S'''
    L = S.shape[0];
    assert L//b >= 2, "Renormalized lattice will have linear dimension <=1"
    newS = np.empty([L//b, L//b],dtype=int)
    for j in np.arange(L//b):
        for i in np.arange(L//b):
            block = S[(b*i):(b*i+b),(b*j):(b*j+b)]
            total = np.sum(block)
            newS[i,j] = assignBlockSpin(total);
```

```
    return newS


### First 7 short range even couplings

def AllEvenCoupling(S):
    '''for spin field config S,
    Integrate measurement of first 7 even correlation functions in one vector'''
    L = S.shape[0];
    assert L >=3, "Lattice too small to fit first 7 even couplings"
    val = np.zeros(7,dtype = float);
    for j in np.arange(L):
        for i in np.arange(L):
            val += [S[i,j]*(S[i,(j+1)%L] + S[(i+1)%L,j]), #nearest neighbor (1,0)
                    S[i,j]*(S[(i+1)%L,(j+1)%L] + S[(i-1)%L,(j+1)%L]), #next nearest neighb
                    S[i,j]*S[i,(j+1)%L]*S[(i+1)%L,(j+1)%L]*S[(i+1)%L,j], # plaquette
                    S[i,j]*(S[i,(j+2)%L] + S[(i+2)%L,j]), #3rd nearest neighbor (2,0)
                    S[i,j]*(S[(i+1)%L,(j+2)%L] + S[(i+2)%L,(j+1)%L]
                            +S[(i-1)%L,(j+2)%L] + S[(i-2)%L,(j+1)%L]),#4th nearest neighbo
                    S[(i+1)%L,j]*S[i,(j+1)%L]*S[(i-1)%L,j]*S[i,(j-1)%L], # sublattice plaq
                    S[i,j]*(S[(i+2)%L,(j+2)%L] + S[(i-2)%L,(j+2)%L])]#5th nearest neighbor
    return val


### First 4 short range odd couplings

def AllOddCoupling(S):
    L = S.shape[0];
    assert L >=3, "Lattice too small to fit first 4 odd couplings"
    val = np.zeros(4,dtype = float);
    for j in np.arange(L):
        for i in np.arange(L):
            val += [0, #magnetization
                    S[i,j]*S[(i+1)%L,j]*S[(i+1)%L,(j+1)%L]+
                    S[i,j]*S[i,(j+1)%L]*S[(i-1)%L,(j+1)%L]+
                    S[i,j]*S[(i-1)%L,j]*S[(i-1)%L,(j-1)%L]+
                    S[i,j]*S[i,(j-1)%L]*S[(i+1)%L,(j-1)%L], #3 spin plaquette
                    S[i,j]*S[(i+1)%L,j]*S[(i+2)%L,(j+1)%L]+
                    S[i,j]*S[i,(j+1)%L]*S[(i-1)%L,(j+2)%L]+
                    S[i,j]*S[(i-1)%L,j]*S[(i-2)%L,(j-1)%L]+
                    S[i,j]*S[i,(j-1)%L]*S[(i+1)%L,(j-2)%L], # 3 spin angle
                    S[i,j]*(S[(i+1)%L,j]*S[(i+2)%L,j] +
                            S[i,(j+1)%L]*S[i,(j+2)%L])] #3 spin row
    val[0] = np.sum(S);
    return val


### Clustering for Wolff Algorithm

def NNBonds(p):
    '''returns set of bonds that connect site p to its 4 nearest neighbors'''
```

```
    i = p[0]; j = p[1];
    nbrs = [(i,(j+1)%L),(i,(j-1)%L),((i+1)%L,j),((i-1)%L,j)]
    bonds = set();
    for n in nbrs:
        bonds.add(frozenset({p,n}))
    return bonds


def buildCluster(S):
    ''''Build Wolff cluster starting from random site for spin configuration S'''
    #random seed location
    L = S.shape[0];
    init = (np.random.choice(L),np.random.choice(L))
    Si = S[init[0],init[1]]

    #cluster starts with 1 element
    cluster = {init}

    #nearest neighbors make up the frontier
    bonds = NNBonds(init)

    #set of points already considered for adding to cluster
    checked = set();

    #while the set of fresh bonds is nonempty, do...
    while (len(bonds) > 0):
        if len(cluster) == Ns:
            break;
        #take out one bond in fresh bond set
        #frozenset ijbond represent unordered edge (i,j)
        ijbond = bonds.pop()
        #add to list of bonds that have been checked
        checked.add(ijbond)
        #pick out element j from edge (i,j)
        jwrap = ijbond - cluster
        #both i and j may already be in cluster, in this case skip to next iteration
        if len(jwrap) == 0:
            continue;
        #otherwise, only i in cluster already, we are left with j
        j = set(jwrap).pop() #convert to usable form
        Sj = S[j[0],j[1]]
        #if parallel to seed spin, activate bond with probability Pij
        if Sj == Si:
            r = np.random.random()
            if r < Pij:
                #add j to cluster, add nearest bonds of j to the fresh bond list
                #also remove bonds already considered from the fresh bond list
                cluster.add(j)
                bonds |= NNBonds(j)
                bonds -= checked
    return cluster
```

```
### Integrated MC + RG simulation function

def Energy(S):
    '''Brute force Find energy of spin configuration S for sanity check'''
    L = S.shape[0];
    E = 0;
    for i in np.arange(L):
        for j in np.arange(L):
            E += K*S[i,j]*(S[i,(j+1)%L] + S[(i+1)%L,j])
    E += h*np.sum(S)
    return E


def RunMCRG(K,h):
    '''Run MCRG simulation to find y_t, y_h exponent,
    keeping Nc_even and Nc_odd coupling terms'''
    print('running MCRG for linear size',L,'lattice.')
    print('Setting K =', K, " and h =",h)

    #measurement accumulators for y_t
    evenK = np.zeros(7,dtype = float)
    evenK_1 = np.zeros(7,dtype = float)
    mix_11 = np.zeros((7,7),dtype=float)
    mix_01 = np.zeros((7,7),dtype=float)
    #measurement accumulators for y_h
    oddK = np.zeros(4,dtype = float)
    oddK_1 = np.zeros(4,dtype = float)
    mix_11_odd = np.zeros((4,4),dtype=float)
    mix_01_odd = np.zeros((4,4),dtype=float)

    #Run simulation
    k = 0;
    for n in np.arange(nmeas+nwarm):
        # Every MC n-loop, build a Wolff cluster and flip it
        # Result: A S-field config drawn with probability propto Boltzmann weight
        cluster = buildCluster(S)
        for p in cluster:
            S[p[0],p[1]] = -S[p[0],p[1]]

        #Sanity checks
        if n % interval == 0:
            energy[k] = Energy(S);
            clustersize[k] = len(cluster);
            k = k+1

        # take measurements every (interval) steps if finished warmup
        if n % interval == 0 and n >= nwarm:
            if n % 100 == 0:
                print("iteration",n)
```

```
            S1 = RGTransform(S,b);
            evenK += AllEvenCoupling(S)
            evenK_1 += AllEvenCoupling(S1)
            oddK += AllOddCoupling(S)
            oddK_1 += AllOddCoupling(S1)
            #A*B = C, B is unknown, A is symmetric
            #Problem: C is not symmetric!?
            mix_11 += np.outer(AllEvenCoupling(S1),AllEvenCoupling(S1))
            mix_01 += np.outer(AllEvenCoupling(S1),AllEvenCoupling(S))
            mix_11_odd += np.outer(AllOddCoupling(S1),AllOddCoupling(S1))
            mix_01_odd += np.outer(AllOddCoupling(S1),AllOddCoupling(S))

    #Results
    evenK /= ndata; evenK_1 /= ndata; mix_11 /= ndata; mix_01 /= ndata;
    oddK /= ndata; oddK_1 /= ndata; mix_11_odd /= ndata; mix_01_odd /= ndata;

    print('evenK = ',evenK)
    print('evenK_1 = ', evenK_1)
    print('mix_11 = ', mix_11)
    print('subtract ',np.outer(evenK_1,evenK_1))
    print('mix_01 = ', mix_01)
    print('subtract ',np.outer(evenK_1,evenK))
    MatA_even = mix_11-np.outer(evenK_1,evenK_1)
    MatC_even = mix_01-np.outer(evenK_1,evenK)
    print('MatA_even (lhs) = ',MatA_even)
    print('MatC_even (rhs) = ',MatC_even)
    print('\n')


    print('oddK = ',oddK)
    print('oddK_1 = ', oddK_1)
    print('mix_11 = ', mix_11_odd)
    print('subtract ', np.outer(oddK_1,oddK_1))
    print('mix_01 = ', mix_01_odd)
    print('subtract ', np.outer(oddK_1,oddK))
    #TODO: cancellation bad, how to avoid?
    MatA_odd = mix_11_odd-np.outer(oddK_1,oddK_1)
    MatC_odd = mix_01_odd-np.outer(oddK_1,oddK)
    print('MatA_odd (lhs) = ',MatA_odd)
    print('MatC_odd (rhs) = ',MatC_odd)
    print('\n')


    return MatA_even, MatC_even, MatA_odd, MatC_odd


def getExponent(MatA,MatC,Nc):
    '''get thermal or magnetic exponent based on output of RunMCRG
    and desired number of coupling constants to consider Nc'''
    LinRGMat = la.solve(MatA[0:Nc,0:Nc],MatC[0:Nc,0:Nc])
    #print('linearized RG transformation = ',LinRGMat)
```

```
    lmbd = la.eigvals(LinRGMat);
    #print('eigenvalues are',lmbd)
    amplitude = np.absolute(lmbd)
    #print('eigenvalue amplitudes are',amplitude)
    #Only the eigenvalue with maximum amplitude is important.
    #This eigenvalue should generically be real
    imax = np.argmax(amplitude)
    y = np.log(lmbd[imax])/np.log(b)
    #print('exponent y = ',y,'\n')
    return y

## Test for specific set of input parameters


# Lattice and MC Parameters, Output File name
L = int(input("Linear Dimension of Lattice: "));
Kc = np.arccosh(3)/4; # Critical temperature Kc assumed to be known
K = Kc; h = 0; #Start on Critical manifold
nwarm = int(input("number of warm up Monte Carlo sweeps:"));
nmeas = int(input("number of measurement Monte Carlo sweeps:"));
interval = int(input("interval between data measurements: "));
filename = input('file name for output data: ')

# RG analysis setting
b = 2;

# Derived constants
Ns = L*L; #total number of grid points
ndata = nmeas//interval
Pij = 1-np.exp(-2*K) # Wolff add probability
energy = np.zeros((nmeas+nwarm)//interval,dtype=float)
clustersize = np.zeros((nmeas+nwarm)//interval,dtype=int)

#Initialize 2d spin field
S = np.random.choice([-1,1],(L,L))


## GO!
MatA_even, MatC_even, MatA_odd, MatC_odd = RunMCRG(K,h)

#plt.plot(energy)
#plt.plot(clustersize)

yt_arr = np.empty(7, dtype = complex);
yh_arr = np.empty(4, dtype = complex);

for i in np.arange(7):
    yt_arr[i] = getExponent(MatA_even,MatC_even,i+1)

for i in np.arange(4):
    yh_arr[i] = getExponent(MatA_odd,MatC_odd,i+1)
```

```
print("mean cluster size as fraction of lattice size: ",np.mean(clustersize)/Ns)
print("y_t array = ", yt_arr)
print("y_h array = ", yh_arr)


#WRITE DATA TO TEXT FILE
f = open("data/"+filename+'.txt','w')
print("================== SETTINGS ==================",file= f)
print("L = ", L, "\n",file = f)
print("K = ", K, "\n",file = f)
print("h = ", h, "\n",file = f)
print("b = ", b, "\n",file = f)
print("nwarm = ", nwarm, "\n",file = f)
print("nmeas = ", nmeas, "\n",file = f)
print("interval = ", interval, "\n",file = f)
print("ndata = ", ndata, "\n",file = f)
print("================== RESULTS ==================", file = f)
print("avg cluster size = ",np.mean(clustersize)/Ns, '*', Ns, "\n",file = f)
print('MatA_even (lhs) = ',MatA_even, "\n",file = f)
print('MatC_even (rhs) = ',MatC_even, "\n",file = f)
print('MatA_odd (lhs) = ',MatA_odd, "\n",file = f)
print('MatC_odd (rhs) = ',MatC_odd, "\n",file = f)
print("y_t array = ", yt_arr ,"\n",file = f)
print("y_h array = ", yh_arr ,"\n",file = f)
f.close()


print("wrote run data to: data/"+filename+'.txt')
```