Peer Assessments (https://class.coursera.org/smac-001/human_grading/)
/ Homework session 8: Cluster sampling, perfect sampling in the Ising model
Help (https://class.coursera.org/smac-001/help/peergrading?url=https%3A%2F%2Fclass.coursera.org%2Fsmac-001%2Fhuman_grading%2Fview%2Fcourses%2F971628%2Fassessments%2F12%2Fsubmissions)

due in 2day 23h

Submission Phase

1. Do assignment ☐ (/smac-001/human_grading/view/courses/971628/assessments/12/submissions)

Evaluation Phase

2. Evaluate peers 🔒 (/smac-001/human_grading/view/courses/971628/assessments/12/peerGradingSets)

Results Phase

3. See results 🔒 (/smac-001/human_grading/view/courses/971628/assessments/12/results/mine)

In this homework session 8 of **Statistical Mechanics: Algorithms and Computations**, you will study the **local Metropolis**, the **heatbath** and the **Cluster Monte Carlo algorithms** for the two-dimensional Ising model on a square lattice with periodic boundary conditions.

At the end of the homework session, you will have validated all three programs against known analytic results. furthermore, you will have compared the results to a classic result of Ferdinand and Fisher, from 1969, analyzed the time-behavior of the cluster algorithm of Wolff (1989), and started to understand Propp and Wilson's coupling from the past approach... So -let's get started with **Homework 8 of Statistical Mechanics: Algorithms and Computations**.

**A** Here, you implement the local Metropolis algorithm, and validate it through the comparison with an exactly known result. In a real-life application, this validation would run for many days on a computer, to establish agreement with the exact result to 5 or 6 significant digits.

**A1** Download (cut-and-paste) the **local Metropolis algorithm** for the Ising model, as shown below.

```
import random, math, os

def energy(S, N, nbr):
    E = 0.0
    for k in range(N):
        E -=  S[k] * sum(S[nn] for nn in nbr[k])
    return 0.5 * E

L = 6
N = L * L
nbr = {i : ((i // L) * L + (i + 1) % L, (i + L) % N,
            (i // L) * L + (i - 1) % L, (i - L) % N) \
                                        for i in range(N)}
T = 2.0
filename = 'local_'+ str(L) + '_' + str(T) + '.txt'
S = [random.choice([1, -1]) for k in range(N)]
nsteps = N * 100
beta = 1.0 / T
Energy = energy(S, N, nbr)
E = []
for step in range(nsteps):
    k = random.randint(0, N - 1)
    delta_E = 2.0 * S[k] * sum(S[nn] for nn in nbr[k])
    if random.uniform(0.0, 1.0) < math.exp(-beta * delta_E):
        S[k] *= -1
        Energy += delta_E
    E.append(Energy)
E_mean = sum(E)/ len(E)
print sum(E) / len(E) / N, 'mean energy'
```

**Greatly increase the number of iterations** of the algorithm. **Run it** for L=6 and for T = 2.0. Check that you recover the exact value E/N =-1.747 (mean energy per spin) (known from exact enumeration). **Communicate the results obtained** in **four independent runs** of the algorithms (no calculation of error bars required). **Indicate your value of nsteps**.

**B** *I* Link <code> Math Edit: Rich Preview

3 600 nsteps, mean energy = -1.72194444444

36 000 nsteps, mean energy = -1.74187037037

360 000 nsteps, mean energy = -1.74417561728

3 600 000 nsteps, mean energy = -1.74959552469

36 000 000 nsteps, mean energy = **-1.74723912037**

So, we do recover the exact value of mean energy per spin (E/N) = -1.747 when we simulate for 36

Attach a file *(supports: txt, png, jpg, gif, pdf, py)*

**A2** Modify your program so that it allows you to read in and write out configurations and also to print configurations on the screen. **For simplicity, we provide this program**, you are free to download (cut-and-paste), run and modify it.

```
import random, math, os, pylab

def x_y(k, L):
    y = k // L
    x = k - y * L
    return x, y

L = 128
N = L * L
nbr = {i : ((i // L) * L + (i + 1) % L, (i + L) % N,
            (i // L) * L + (i - 1) % L, (i - L) % N) \
                                    for i in range(N)}

T = 1.0
filename = 'local_'+ str(L) + '_' + str(T) + '.txt'
if os.path.isfile(filename):
    f = open(filename, 'r')
    S = []
    for line in f:
        S.append(int(line))
    f.close()
    print 'starting from file', filename
else:
    S = [random.choice([1, -1]) for k in range(N)]
    print 'starting from scratch'
nsteps = N * 100
beta = 1.0 / T
for step in range(nsteps):
    k = random.randint(0, N - 1)
    delta_E = 2.0 * S[k] * sum(S[nn] for nn in nbr[k])
    if random.uniform(0.0, 1.0) < math.exp(-beta * delta_E):
        S[k] *= -1

conf = [[0 for x in range(L)] for y in range(L)]
for k in range(N):
    x, y = x_y(k, L)
    conf[x][y] = S[k]
pylab.imshow(conf, extent = [0, L, 0, L], interpolation='nearest')
pylab.set_cmap('hot')
pylab.colorbar()
pylab.title('Local_'+ str(T) + '_' + str(L))
pylab.savefig('Local_'+ str(T) + '_' + str(L)+ '.png')
pylab.show()
f = open(filename, 'w')
for a in S:
   f.write(str(a) + '\n')
f.close()
```

From a random initial configuration, run this program at high temperature **T = 3.0  (L=128)** and at the critical temperature **T_crit = 2.27 (L=128)**. Run sufficiently long so that you have reached what you would consider as the t-> \infty limit. (At T_crit, you should run your code for at least a few minutes). **Upload**

**typical configurations that you obtained and comment on what you see.** Notice that subsequent runs of this program realize a Markov chain (three runs with 100 + 100 + 100 iterations = 1 run with 300 iterations).

At low temperature, **T = 1.0**, run your code for **L=32,** first for **10 N** iterations, then again, **several times for 10 N** iterations, then several times **for 100 N** iterations (if you see a stripe, run your program longer, up to several ties 1000 N or several times 10000 N iterations). You should observe that, after a quite long time, your simulation goes from a state with domain walls into a purely ferromagnetic state (essentially all spins +1 or essentially all spins -1). **Upload two or three graphics files at different times. Do you observe** (running for several times 10000 N iterations) that the local Metropolis algorithm flips between configurations of negative overall magnetization (mostly black) and configurations of positive overall magnetization (mostly white)?

At low temperature, **T = 1.0** (as before), run your code for **L=128** and **upload a graphics file**.
**Explain** what you observe. In particular, do you observe that the system becomes homogeneous (mostly white or mostly black) on the available time scales?

B *I* Link <code> Math Edit: Rich Preview

Attach a file *(supports: txt, png, jpg, gif, pdf, py)*

**B** In this section, you run the Wolff cluster algorithm, one of the great (but little understood) algorithms of statistical physics.

**B1** Download (cut-and-paste) the Wolff cluster Monte Carlo algorithm for the Ising model shown below. Familiarize yourself with how it works before going on. **Explain in a few words** what the **Pocket** stands for.

```
import random, math

def energy(S, N, nbr):
    E = 0.0
    for k in range(N):
        E -=  S[k] * sum(S[nn] for nn in nbr[k])
    return 0.5 * E

L = 6
N = L * L
nbr = {i : ((i // L) * L + (i + 1) % L, (i + L) % N,
            (i // L) * L + (i - 1) % L, (i - L) % N)
                                        for i in range(N)}
T = 2.0
p  = 1.0 - math.exp(-2.0 / T)
nsteps = 10000
S = [random.choice([1, -1]) for k in range(N)]
E = [energy(S, N, nbr)]
for step in range(nsteps):
    k = random.randint(0, N - 1)
    Pocket, Cluster = [k], [k]
    while Pocket != []:
        j = random.choice(Pocket)
        for l in nbr[j]:
            if S[l] == S[j] and l not in Cluster \
                   and random.uniform(0.0, 1.0) < p:
                Pocket.append(l)
                Cluster.append(l)
        Pocket.remove(j)
    for j in Cluster:
        S[j] *= -1
    E.append(energy(S, N, nbr))
print sum(E)/ len(E) / N
```

Greatly increase the number of iterations of the algorithm. Run it for **L=6 and T = 2.0**. Check that you recover the exact value for the mean energy **E/N(T=2, L=6) = -1.747** (known from exact enumeration). **Communicate the results obtained** in four independent runs of the algorithms. Do not forget to **indicate your value of nsteps**.
**Next, answer the following question**: is it necessary to **pick** a **random element j** of the pocket (element that we eliminate from the pocket through **Pocket.remove(j)**), or can we simply pop the last element of Pocket (j = Pocket.pop())?
**Modify then run** the modified program (note that it has one less line, the Pocket.remove() line is no longer needed), and **explain your observations**.

**B** *I* Link <code> Math Edit: Rich Preview

Attach a file (supports: txt, png, jpg, gif, pdf, py)

**B2** As discussed in lecture 8, the **Ising model was solved exactly by Onsager**, in 1944, for the infinite lattice. The exact solution for the finite lattice was obtained by **Kaufman in 1949** . Based on her (sic!) magnificent paper, **Ferdinand and Fisher, in 1969**, computed the specific heat for finite lattices with periodic boundary conditions, exactly the system that we consider, and the result is shown here:

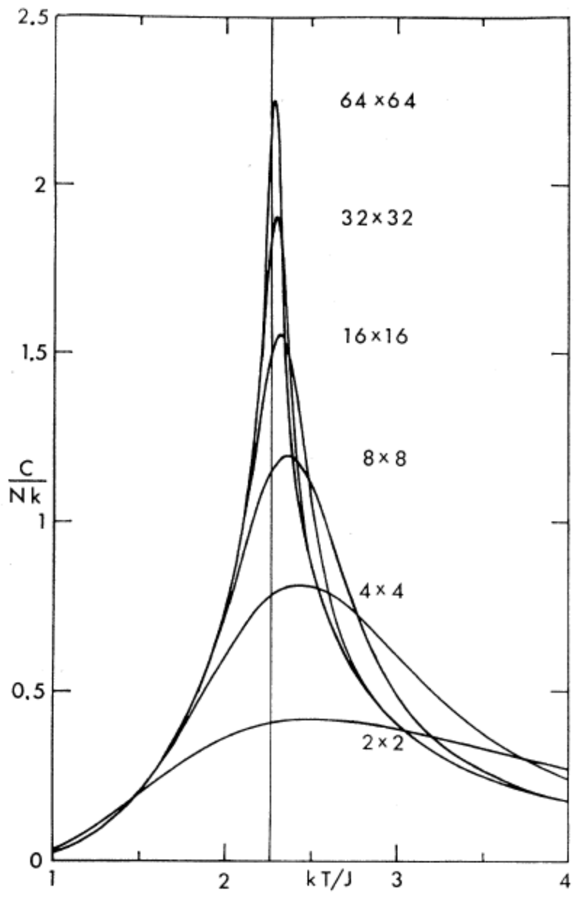


FIG. 1. The specific heat per spin for small Ising lattices; exact results for the $m \times n$ square lattice with periodic boundary conditions are displayed for $m=n=2, 4, 8, 16, 32$, and 64 ($N=mn$). The limiting critical point is marked by a vertical line.

Now, incorporate into the cluster algorithm the read-in, write-out part of section **A2**, which allows you to do a 'warm start' (that is, discard some initial data). Run this modified cluster algorithm for lattices of size L=2, 4, 8, 16, 32 and check Ferdinand and Fisher's analytical results. To do so, **implement a few lines analogous to:**

```
E_mean = sum(E)/ len(E)
E2_mean = sum(a ** 2 for a in E) / len(E)
cv = (E2_mean - E_mean ** 2 ) / N / T ** 2
```

(**Remember the definition** of the specific heat from lecture 8). **Communicate your results** (without error bars, just quote the results of one or two runs, after a warm start, for each value of the parameters) at the critical temperature T = 2.27. NB: we just need rough agreement.

**Can you confirm that**, as indicated in the conclusion of lecture 8, observables as the specific heat for the Ising model in the transition region strongly depend on lattice size?

B I Link <code> Math Edit: Rich Preview

Attach a file *(supports: txt, png, jpg, gif, pdf, py)*

**C** Here, you run the heatbath algorithm.

**C1** Download (cut-and-paste) the heatbath algorithm for the Ising model shown below. Familiarize yourself with how it works before going on.

```
import random, math

L = 6
N = L * L
nbr = {i : ((i // L) * L + (i + 1) % L, (i + L) % N,
            (i // L) * L + (i - 1) % L, (i - L) % N) \
                                    for i in range(N)}
nsteps = 10 * N
T = 2
beta = 1.0 / T
S = [random.choice([-1, 1]) for site in range(N)]
E = -0.5 * sum(S[k] * sum(S[nn] for nn in nbr[k]) \
                                for k in range(N))
Energies = []
for step in range(nsteps):
    k = random.randint(0, N - 1)
    Upsilon = random.uniform(0.0, 1.0)
    h = sum(S[nn] for nn in nbr[k])
    Sk_old = S[k]
    S[k] = -1
    if Upsilon < 1.0 / (1.0 + math.exp(-2.0 * beta * h)):
        S[k] = 1
    if S[k] != Sk_old:
        E -= 2.0 * h * S[k]
    Energies.append(E)
print sum(Energies) / len(Energies) / N
```

Greatly increase the number of iterations of the algorithm. Run it for L=6 and for T = 2.0. Check that you recover the exact value for the mean energy E/N(T=2, L=6) = -1.747 (known from exact enumeration). **Communicate the results obtained** in four independent runs of the algorithm. **Indicate your value of nsteps**.

**B** *I* Link <code> Math Edit: Rich Preview

Attach a file *(supports: txt, png, jpg, gif, pdf, py)*

**C2** Modify the heatbath algorithm so that it does TWO computations in parallel, as discussed in Tutorial 8: one starting from the all plus spin configuration, and one from the all minus configuration. **For simplicity, we provide this program**, you are free to download (cut-and-paste), run and modify it, but familiarize yourself thoroughly with this program.

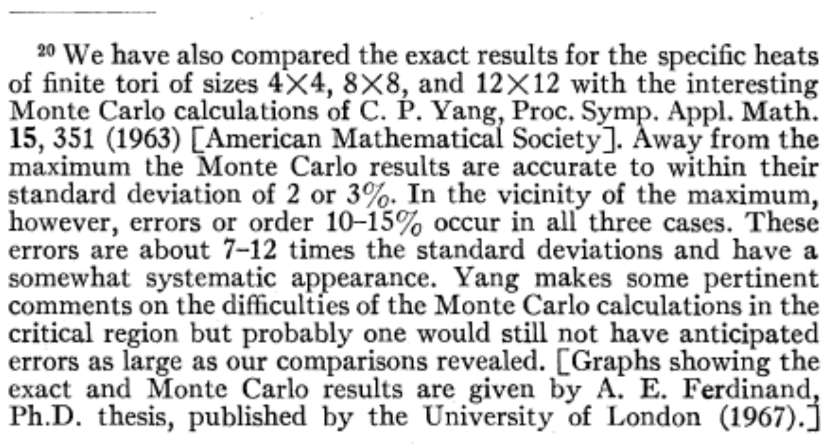
```
import random, math

L = 32
N = L * L
nbr = {i : ((i // L) * L + (i + 1) % L, (i + L) % N,
            (i // L) * L + (i - 1) % L, (i - L) % N) \
                                    for i in range(N)}
T = 2.3
beta = 1.0 / T

t_coup = []
for iter in range(10):
    print iter
    S0 = [1] * N
    S1 = [-1] * N
    step = 0
    while True:
        step += 1
        k = random.randint(0, N - 1)
        Upsilon = random.uniform(0.0, 1.0)
        h = sum(S0[nn] for nn in nbr[k])
        S0[k] = -1
        if Upsilon < 1.0 / (1.0 + math.exp(-2.0 * beta * h)):
            S0[k] = 1
        h = sum(S1[nn] for nn in nbr[k])
        S1[k] = -1
        if Upsilon < 1.0 / (1.0 + math.exp(-2.0 * beta * h)):
            S1[k] = 1
        if step % N == 0:
            n_diff = sum(abs(S0[i] - S1[i]) for i in range(N))
            if n_diff == 0:
                t_coup.append(step / N)
                break
print t_coup
print sum(t_coup) / len(t_coup)
```

Compute the time (in "**sweeps**", that is in number of steps / N) at which the algorithm couples, that is, at which the difference between the configurations generated falls to zero. Plot this coupling time (averaged over 10 runs with different random numbers) for L = 32 at temperature T= 5.0, 4.0, 3.0, 2.5, 2.4, 2.3,**Attention**: run T=2.3 only if you have a lot of CPU time available. **Describe** what you see. In the Ising model, the coupling time is, up to a logarithmic factor in N, equal to the correlation time, so this coupling time informs you of the order of magnitude of the correlation time (as discussed in Tutorial 1).

Ferdinand and Fisher, in a footnote of their 1969 paper that we discussed earlier, compared with their exact calculations with the earliest Monte Carlo calculations on the Ising model. Here is the footnote:


[20] We have also compared the exact results for the specific heats of finite tori of sizes 4×4, 8×8, and 12×12 with the interesting Monte Carlo calculations of C. P. Yang, Proc. Symp. Appl. Math. **15**, 351 (1963) [American Mathematical Society]. Away from the maximum the Monte Carlo results are accurate to within their standard deviation of 2 or 3%. In the vicinity of the maximum, however, errors or order 10–15% occur in all three cases. These errors are about 7–12 times the standard deviations and have a somewhat systematic appearance. Yang makes some pertinent comments on the difficulties of the Monte Carlo calculations in the critical region but probably one would still not have anticipated errors as large as our comparisons revealed. [Graphs showing the exact and Monte Carlo results are given by A. E. Ferdinand, Ph.D. thesis, published by the University of London (1967).]

NB: "Torus" is the same as "lattice with periodic boundary conditions".

In the light of your exact bounds on the correlation time, can you **explain the insufficient agreement between theory and numerics reported by Ferdinand and Fisher**? (Remember the lesson of homework 1: underestimating the correlation time makes one underestimate the error).

B I Link <code> Math Edit: Rich Preview

Attach a file *(supports: txt, png, jpg, gif, pdf, py)*

The algorithm in C2 couples, and its coupling time provides a rigorous limit on the convergence time. Nevertheless, the configuration obtained is not really a random configuration, simply because the coupling time itself has fluctuations.

For your information and enjoyment, we provide here a rudimentary yet exact algorithm following the "coupling from the past principle" that, rather than simulating from time t=0 to a coupling time t_coup:

```
import random, math

L = 16
N = L * L
nbr = {i : ((i // L) * L + (i + 1) % L, (i + L) % N,
            (i // L) * L + (i - 1) % L, (i - L) % N) \
                                    for i in range(N)}
nsteps = 10
T = 3.0
beta = 1.0 / T
S0 = [1] * N
S1 = [-1] * N
k = {}
Upsilon = {}
n_diff = 10
while n_diff != 0:
    nsteps *= 2
    for step in range(-nsteps, 0):
        if step not in k:
            k[step] = random.randint(0, N - 1)
            Upsilon[step] = random.uniform(0.0, 1.0)
        h = sum(S0[nn] for nn in nbr[k[step]])
        S0[k[step]] = -1
        if Upsilon[step] < 1.0 / (1.0 + math.exp(-2.0 * beta * h)):
            S0[k[step]] = 1
        h = sum(S1[nn] for nn in nbr[k[step]])
        S1[k[step]] = -1
        if Upsilon[step] < 1.0 / (1.0 + math.exp(-2.0 * beta * h)):
            S1[k[step]] = 1
    n_diff = 0
    for i in range(N):
        if S0[i] != S1[i]: n_diff += 1
    print nsteps, n_diff
```

The configuration at t=0 is a perfect sample, following the coupling from the past idea of Propp and Wilson (1996). You can see that an infinite simulation, from t = -infty up to t = 0, would output the sample at t=0.

**NB:** This part is for your personal enchantment, no points given, no questions asked.

B *I* Link <code> Math Edit: Rich Preview

Attach a file *(supports: txt, png, jpg, gif, pdf, py)*

NOTE: You have saved, *but not submitted*, your work. If you do not submit your work before the deadline, you will not be allowed to evaluate the work of your peers and your own work will not be evaluated. **Don't wait until the last minute**, in case you have unforeseen problems such as power outages. Note that after you submit, you can still make changes and re-submit as long as it's before the deadline -- so **submit early**.

☐ In accordance with the Honor Code, I certify that my answers here are my own work, and that I have appropriately acknowledged all external sources (if any) that were used in this work.

Save draft Submit for grading