Monte Carlo Exercise Session

Helmut G. Katzgraber and Andrew J. Ochoa
Department of Physics and Astronomy, Texas A&M University

---

01. Using and testing random number generators — The goal of this problem is to familiarize yourself with the use of a random number generator and routines to automatically seed the generator. You can access the necessary C files at this URL

```
http://katzgraber.org/outgoing/BAD-HO/01
```

The above generator is known as r1279 and is a Lagged Fibonacci generator. If you program in C++, please use the Mersenne Twister either from

```
http://goo.gl/hfm6ZF
```

or the GNU Scientific Library [`http://www.gnu.org/software/gsl`]. The generator `helmutrand` will be used later.

Note: You are welcome to use the provided C code as a guide and then solve the problem in a language of your choice. We have produced Python versions of the code which you can also find under the above link. If, however, you program in some other language, translating `helmutrand` to a different language should be easily accomplished.

a) Based on the example you downloaded from the class website, write a program that calls `r1279` (or the Mersenne Twister) $N = 10^k$ times with $k_{\max} = 8$ and averages over random floats in the interval $[0, 1]$. The seed should be automatically generated. The exact value for the average is 0.5. Show how the estimate improves when N is increased in powers of 10 and that the error (deviation from exact value) is roughly proportional to $\sqrt{N}$. You will see strong fluctuations in the data, which is normal. If you want to reduce these, you can average your results for each k over approximately 10 independent runs to improve the statistics. However, this is not necessary.

b) Repeat the study done in section (a). However, this time use the generator `helmutrand`. Also plot the deviation from the exact value. What do you observe? Do you conclude that these random numbers are good?

c) 3D spectral test: Change the routines you developed in section (b) such that they print out 10000 triplets of consecutive random numbers. Store these data in a file and visualize them using, for example, gnuplot with the following command

```
gnuplot> splot 'result.dat' w points pt 0
```

You should be able to rotate the image in space. See if you can find any correlations. Store a image of your result.

---

02. Simple-sampling estimate of π — The goal of this problem is to illustrate how $\pi = 3.1415\ldots$ can be computed by random sampling of the unit disk. Starting from the pseudo-code presented in class, write a program that calculates π.

In your simulation run the code multiple times for $N = 10^k$, $k = 1, \ldots 8$ random numbers. See how the estimate for π improves with increasing N and compute the deviation from the exact result, i.e., $\varepsilon = |\pi - \pi_{\text{estimate}}|$. Perform a log-log plot of the error as a function of N and show that the data can be fit to a straight line of slope $-1/2$. To improve statistics, for each N average over $M = 100$ runs.

---

03. Simple-sampling Monte Carlo integration — The goal of the exercise is to apply the concepts learned in problem 2 to the function $f(x) = x^n$ where the integral is exactly known, namely $I[f(x)] = 1/(n + 1)$ in the interval $[0, 1]$. Set $n = 3$, you can change the value of the exponent n later if you wish. Start again from the pseudo-code presented in class.

As in problem 2, run the code multiple times for $N = 10^k$, $k = 1, \ldots 8$ random numbers. See how the estimate for $I[f(x)]$ in the interval $[0, 1]$ improves with increasing N and compute the deviation from the exact result, i.e., $\varepsilon = |I - I_{\text{estimate}}|$. Again, the error should scale $\sim N^{-1/2}$.

---

04. Estimating π using Markov-chain Monte Carlo — The goal of this problem is to illustrate how π can be computed by random sampling of the unit disk, however, using a Markov chain. For the sake of simplicity, we shall neglect autocorrelation effects that in this case would only influence the error. Therefore, measurements should be done at every step of the simulation. Starting from the pseudo-code presented in class, modify your program to calculate π using a Markov chain.

In your simulation run the code multiple times for $N = 10^k$, $k = 1, \ldots 8$ random numbers. See how the estimate for π improves with increasing N and compute the deviation from the exact result, i.e., $\varepsilon = |\pi - \pi_{\text{estimate}}|$. Perform a log-log plot of the error as a function of N and show that the data can be fit to a straight line of slope $-1/2$. To improve statistics, for each N average over $M = 10$ runs.

Note that for the Markov chain you will have to select a step size in the interval $[-p, p]$. To do so, take a uniform random number x and compute a shifted uniform random number in the interval $[a, b]$ via $y = a + (b - a)x$. The value of p strongly influences the algorithm. If p is too small, then it will converge slowly. If p is too large, many moves will be rejected. Ideally, 50% of the moves should be accepted. To verify this, measure the probability P for a move to occur, i.e., $P = A/N$, where A is the number of accepted moves. Values of p between 0.2 and 1.0 seem to be optimal.

05. Importance sampling Monte Carlo estimate of an integral — The goal of this exercise is to show that with the same numerical effort as in exercise 3 importance sampling delivers smaller statistical errors. We want to compute the integral of the function $f(x) = x^n$ in the interval $[0, 1]$, but instead of using uniform random numbers, we want to use power-law distributed random numbers according to the distribution $p(x) = (k + 1)x^k$ with $k < n$. Power-law distributed random number can be computed from uniform random numbers by transforming a uniform random number x in the following way:

$$y = x^{1/(k+1)}$$

with x in $[0, 1]$ uniform. The importance-sampling estimate of the integral $I[f(x)]$ is then given by a Monte Carlo sampling where $f(x)$ is replaced by the following function $f(x) \to f(y)/p(y)$ with $f(y) = y^n$ and $p(y) = (k + 1)y^k$ and y power-law distributed according to $p(y)$. Start with $n = 3$ ($I = 0.25$) and $k = 2.5$ and compare to the results obtained in problem 3. As in problem 3, run the code multiple times for $N = 10^m$, $m = 1, \ldots 8$ random numbers. See how the estimate for $I[f(x)]$ improves with increasing N and compute the deviation from the exact result, i.e., $\varepsilon = |I - I_{\text{estimate}}|$. Again, the error should scale $\sim N^{-1/2}$. Plot your result with the error estimates you obtained in 3 and compare.

06. One-dimensional random walk (optional) — Consider a random walker who starts at $y = 0$ and walks along a line along the y-axis. At each time step $t = 1, 2, 3, \ldots$ the walker moves either one step up or one step down with equal probability. By averaging over a sufficiently large number of walks M, show numerically that $\langle y(t) \rangle \approx 0$ and that $\langle y^2(t) \rangle \sim t$. The average $\langle \cdots \rangle$ is over walkers. Plot both $\langle y(t) \rangle$ and $\langle y^2(t) \rangle$ vs t for $t_{\max} \sim 1000$. Hint: While the random walk is for 1000 steps, store data only every 10th step to save disk space. Furthermore, average the data over 10^4 runs to estimate statistical error bars.

07. The one-dimensional Ising model — The goal of the problem is to simulate a one-dimensional Ising chain and compute the energy per spin E/N as a function of temperature T in the range $[0.5, 5.0]$ for the model and compare to the exact analytical expression computed by Ernst Ising in his PhD thesis:

$$\frac{E}{N} = \left(\frac{e^{-J/kT} - e^{J/kT}}{e^{-J/kT} + e^{J/kT}} \right) . \tag{1}$$

The one-dimensional Ising model is described by the Hamiltonian

$$\mathcal{H} = -J \sum_{i=1}^{N} S_i S_{i+1} . \tag{2}$$

In your simulations set the energy scale to 1, i.e., $J = 1$. It is recommended you code the problem in a bottom-up fashion, i.e., start with the core routines and then put everything together in one program.

To reduce finite-size effects, we place the spins on a one-dimensional ring. This means that spin S_i is connected by an interaction $-J$ to spin S_{i+1} and, in particular S_N connects to S_1. The easiest (and most generic) way to implement this such that you can use the code later in higher space dimensions is to store all N spins in a one-dimensional array `spins[i]`. In order to figure out who is neighbor of who, you should also implement a lookup table `nb[j][i]` that returns the `j`th neighbor of spin `i` when called. A simple way to accomplish this can be done with the following code:

```
for (i = 1; i <= N; i++) {          /* loop over all spins */
    if (i == 1) {                   /* if first spin, left nb is Nth */
        nb[1][i] = N;
        nb[2][i] = i + 1;
    }
    else if (i == N){               /* if last spin, right nb is 1 */
        nb[1][i] = i - 1;
        nb[2][i] = 1;
    }
    else {                          /* any other spin in the middle */
        nb[1][i] = i - 1;
        nb[2][i] = i + 1;
    }
}
```

In one space dimension each spin has two neighbors: left [1] and right [2]. The matrix `nb[i][j]` is thus a $2 \times N$–dimensional object. Note that this lookup table can be trivially extended to higher space dimensions.

For the Monte Carlo update routine you need to compute the difference in energy between the old and new configuration when flipping a randomly-selected spin S_i. this means, the change in energy will be $\Delta E = 2S_i(S_{i-1} + S_{i+1})$. To flip a spin, simply use the Metropolis algorithm:

```
s = irand(1,N);  /* select a random spin index in (1,N) */

delta = 0;

for(j = 1; j <= 2; j++){
    delta += 2*spins[s]*spins[nb_matrix[j][s]];
}

if (rand() < (exp(-1.0*delta/temp))){
    spins[s]  = -spins[s];
    *energy  += ((double) delta);
}
```

Note that it is *very* important in one space dimension to select the spin at random. We call N spin updates one *lattice sweep*.

a) Write the Monte Carlo code to simulate the one-dimensional Ising model. Thermalize for the first 3000 lattice sweeps and measure over the subsequent 3000 lattice sweeps. Do not worry about autocorrelation effects. Measure the energy during the measurement phase at each sweep and for $T = 0.50, 1.00, 1.50, \ldots, 5.00$. Average over 10 runs. Measure the average energy, as well as the average of the squared energy to compute a statistical error bar (averaged over the aforementioned 10 runs). Run your simulation for $N = 1000$ spins and show in a figure that the result converges for increasing N to the exact expression by Ernst Ising.

b) In part (a) you were asked to thermalize your simulations with 3000 lattice sweeps. This problem will explore the validity of that thermalization. Run a simulation with 100 independent Markov chains using your code. Have this simulation run for 6000 lattice sweeps at $T = 0.5$, measuring $\langle E \rangle / N$ and $\langle M^2 \rangle / N$ for each sweep, where $\langle \ldots \rangle$ indicates an average over Markov chains. Plot these observables versus the number of sweeps in separate plots, describe how you can tell that the simulation has equilibrated, and comment on whether or not the choice of 3000 thermalization sweeps was valid.

---

08. The 2D Ising Model (optional) — The 2D Ising model has more interesting physics and so is more commonly studied. In this problem, you will test the modularity of your program and explore the 2D Ising model. To simplify things, it is worth noting that a 1D array can be used like a 2D matrix by changing the indices. For example, if you have an $L \times L$ matrix M, and want to store in a 1D array A, then $M[i][j] = A[(i-1)L + j]$ if $i, j = 1 \ldots L$, and $M[i][j] = A[iL + j]$ if $i, j = 0 \ldots L - 1$. Other things to note during your conversion are that each spin now has 4 neighbors and the ground state energy has changed, i.e., $E_0/N = -2J$.

Modify your program from problem 7 to simulate a 2D Ising model with periodic boundary conditions. In 2D, $N = L^2$, where L is the side length of the system. Repeat all calculations from problem 7 in two space dimensions, but now for the temperatures $T = 2.0 \ldots 2.5$ with $\Delta T = 0.02$ and for system sizes $L = 8$, 16, and 32.

---

09. Glassy Systems — This problem will explore why glassy systems are computationally hard. Consider a 1D Ising spin glass described by the Hamiltonian

$$\mathcal{H} = -J \sum_{i=1}^{N} S_i S_{i+1} \; , \tag{3}$$

where $J_{ij} = \pm 1$ random with equal probability. What this means is that in your code for the 1D Ising model, you need to randomly assign positive an negative values to the interactions with equal probability.

Run a simulation with 1000 independent Markov chains of a $N = 100$ system (each chain having a different set of J_{ij}!) at $T = 0.4$. Plot $\langle E \rangle$ versus Monte Carlo sweep number for up to 40000 sweeps and show that glassy systems thermalize much slower than ferromagnets. Optionally, repeat this simulation in 2D with $N = 8^2$.

---

10. Solving the Traveling Salesman Problem with Simulated Annealing — In this exercise we optimize a pre-defined tour of 72 cities using simulated annealing. The tour can be downloaded from

```
http://katzgraber.org/outgoing/BAD-HO/10
```

Each row lists the x and y coordinates of a city. If you want to visualize the tour, simply type in gnuplot

```
gnuplot > plot 'tour.txt' w l , 'tour.txt' t ''
```

The goal is to find the shortest tour between the cities. Start by reading in the pre-defined cities into your program. The cities are on a 200×200 integer grid. The distance between two cities c_1 and c_2 is given by their Euclidean distance, i.e.,

$$d(1,2) = |c_1 - c_2| = \sqrt{(c_1^x - c_2^x)^2 + (c_1^y - c_2^y)^2}$$

The "cost function" for the problem is the length of the tour. This means you will have to calculate the sum of all distances, i.e.,

$$\ell = d(1,N) + \sum_{i=1}^{N-1} d(i,i+1)$$

Note that the first term in the definition of ℓ closes the tour, i.e., connects the first with the last city. The main update routine then picks two different cities on the current tour (make sure these are random and distinct!) and proposed a new tour by swapping these. You can then compute the "energy" of the proposed swap via

```
energy = length(newtour) - length(originaltour)
```

and then propose a Monte Carlo update:

```
if(energy < 0){
    originaltour <- newtour
}
else if (rand() < (exp(-1.0*energy/T))){
    originaltour <- newtour
}
```

After attempting the Monte Carlo update, make sure you keep track of the optimal tour, i.e.,

```
if(length(originaltour) < length(mintour)){
    mintour <- originaltour
}
```

Note: When you read in the initial tour from the file, make sure you copy it to `mintour` for the initialization. In the main routine of your program you will run the annealing schedule. Start with $T_{\max} = 20.0$ and decrease the temperature in a linear schedule with $\Delta T = 0.1$ in 200 steps to $T = 0$. At the first temperature (20.0) run 10 000 Monte Carlo updates. After each step in the temperature, increase the number of Monte Carlo updates linearly, i.e.,

```
for(j = 1; j <= 200; j++){

    T = T - 0.1;                    /* linearly decrease temperature */
    sweeps = 10000*j;               /* linearly increase sweeps      */

    for(m = 1; m <= sweeps; m++){
        update(tour,mintour,T);     /* main update routine           */
    }
}
```

a) Write a program to simulate the TSP with simulated annealing. Make sure your program stores the minimum length obtained as a function of the annealing temperature. Furthermore, make sure your program stores the optimal tour.

b) Plot the minimum length as a function of temperature and thus show that you are optimizing the tour, as well as the optimal tour.

Note: You might have to run your program multiple times. Recall that simulated annealing can become stuck in a metastable valley of the energy landscape (i.e., not the optimum). Your optimal tour should be shorter than ~ 1340 length units. With the given choice of parameters, you will see that you can obtain tour lengths as long as 1400 length units if you happen to pick the wrong seed. This is why it is so important to do multiple runs with simulated annealing.