Monte Carlo Methods Exercises – Sessions 1 & 2

Introduction – The goal of these exercise sessions is to better understand the concepts taught in class, such as Monte Carlo integration, simple versus importance sampling, Markov chains, as well as simulations of the Ising model. You can download all material for the exercises from

http://katzgraber.org/teaching/SUM/2012-bad_honnef

The material contains example programs which are the solutions to the problems listed below. Please first attempt to write your own code. If you have problems, feel free to look at and use the example code written in C. Pseudo-code for the different routines/exercises can be found at

http://arxiv.org/pdf/0905.1629.pdf

You are not expected to complete all problems in the first part of the exercises. I do, however, expect you to complete at least problems 1 – 4. The last three are more complex and should be completed in the second part of the exercise session.

Note: Compiling and running code – All codes are written in C. They can be edited with any simple editor (vi, pico, nano, emacs, TextEdit...) and compiled with the added Makefile (just type 'make' in a Unix shell) or using 'gcc *.c'. If you need help with the Unix environment, please inform the instructor immediately. The codes also compile and run on Windows.

Problem 1: Using a random-number generator in C – The goal of this problem is to familiarize yourself with the use of a random number generator and routines to automatically seed the generator. Please download 'problem_1_using_rng' from the exercise website. The directory contains C source code for the r1279( ) random number generator, as well as routines (seedgen.c) to seed the generator. (You will need this for problems 2 – 6 if you program in C).

Write a program that generates 10^N ($N_{\max} = 6$ is enough) uniform random numbers in the interval [0,1] and averages these. The seed should be automatically generated. The exact value for the average should be 0.5. Study how the average changes when N is increased.

If you do not program in C or know/use a better generator, please feel free to use that for this exercise, as well as the remaining problems.

Note: the program seedgen.c takes the process ID (PID), as well as the wall-clock-time and produces a semi-random seed via a linear-congruential-like operation. Therefore, every time you run the code, it should produce a *different* output.

Problem 2: Estimating π using simple-sampling Monte Carlo – The goal of this problem is to illustrate how π = 3.1415... 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 $N = 10^i$, $i = 1, 2, 3, ...$ random numbers. See how the estimate for π improves with increasing N and compute the deviation from the exact result: error = $|\pi - \pi_{\text{estimate}}|$.

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.

```
algorithm simple_pi
    initialize n_hits     0
    initialize m_trials   10000
    initialize counter    0

    while(counter < m_trials) do
        x = rand(0,1)
        y = rand(0,1)
        if(x**2 + y**2 < 1)
            n_hits++
        fi
        counter++
    done

return pi = 4*n_hits/m_trials
```

Problem 3: Simple-sampling Monte Carlo estimate of the integral $f(x) = x^n$ – The goal of the exercise is to apply the concepts learned in problem 2 and apply these to a real function where the integral is exactly known, namely $I[f(x)] = 1/(n + 1)$ in the interval [0,1]. For now, set $n = 3$, you can change the value of the exponent n later. Start from the pseudo-code presented in class.

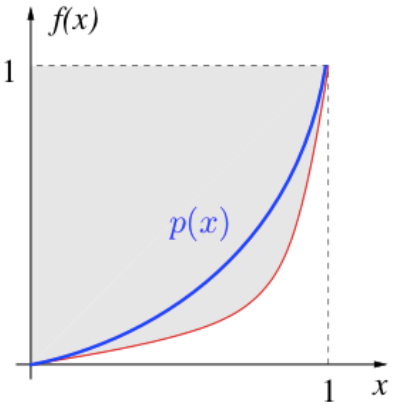
As in problem 2, run the code multiple times for $N = 10^i$, $i = 1, 2, 3, ...$ random numbers. See how the estimate for $I[f(x)]$ in the interval [0,1] improves with increasing N and compute the deviation from the exact result: error = $|I - I_{\text{estimate}}|$. Again, the error should scale ~ $N^{-1/2}$.

```
algorithm simple_integrate
    initialize integral   0
    initialize m_trials   10000
    initialize counter    0

    while(counter < m_trials) do
        x = rand(0,1)
        integral += x**n
        counter++
    done

return integral/m_trials
```

Problem 4: Importance-sampling Monte Carlo estimate of the integral $f(x) = x^n$ – The goal of this exercise is to show that with the *same* numerical effort as in problem 3 importance sampling delivers smaller 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:

$$y = x^{1/(k+1)} \qquad \text{with } x \text{ in [0,1] uniform}$$



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

$$f(x) \longrightarrow f(y)/p(y)$$

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 $N = 10^i$, $i = 1, 2, 3, ...$ random numbers. See how the estimate for $I[f(x)]$ improves with increasing N and compute the deviation from the exact result: error = $|I - I_{\text{estimate}}|$. Compare to the results of problem 3.

Problem 5: Esimating 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 will 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 $N = 10^i$, $i = 1, 2, 3, ...$ random numbers. See how the estimate for π improves with increasing N and compute the deviation from the exact result: error = $|\pi - \pi_{\text{estimate}}|$.

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 for a move to occur, i.e., prob = (accepted_moves/total_attempts). Values of p between 0.2 and 1.0 seem to be optimal.

```
algorithm markov_pi
    initialize n_hits     0
    initialize m_trials   10000
    initialize x          0
    initialize y          0
    initialize counter    0

    while(counter < m_trials) do
        dx = rand(-p,p)
        dy = rand(-p,p)
        if(|x + dx| < 1 and |y + dy| < 1)
            x = x + dx
            y = y + dy
        fi
        if(x**2 + y**2 < 1)
            n_hits++
        fi
        counter++
    done

return pi = 4*n_hits/m_trials
```

Problem 6: Simulating the one-dimensional Ising model – In this problem we now apply the concepts learned to the simulation of a physical system, the one-dimensional Ising model on a closed ring. This problem is slightly more work intensive.

The goal of the problem is to compute the energy per spin $e = E/N$ as a function of temperature T in [0.2, 5.0] for the model and compare to the exact analytical expression computed by Ernst Ising in his thesis:

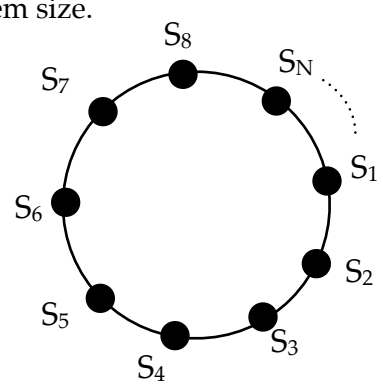
$$e = \frac{1}{N}\left[\frac{e^{-J/T} - e^{J/T}}{e^{-J/T} + e^{J/T}}\right]$$

Run your code for ~3000 equilibration Monte Carlo sweeps (N attempted spin updates) and then for 3000 measurement Monte Carlo sweeps and record the internal energy. Compare your results for N = 3, 10, 30, 100, 300 spins to the exact expression. You should see convergence to the exact values [also provided as a text file] for increasing system size.

The Hamiltonian of the 1D Ising model is given by

$$\mathcal{H} = -J \sum_i S_i S_{i+1} ,$$

where the spins lie on a one-dimensional chain with periodic boundaries. For simplicity, we set J = 1. It is recommended to tackle the problem bottom-up, i.e., write individual routines first.



update.c: 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: $dE = 2S_i(S_{i-1} + S_{i+1})$. To flip a spin

```
if(r1279(  ) < exp(-dE/T)){
   flip spin;
   update energy;
}
```

The function ir1279range(1,N) will select a random spin in the interval [1,N]. Keep in mind that we use periodic boundaries, i.e., the 'left' neighbor if spin S_1 is the spin S_N. If this is unclear, please look at the provided code.

Problem 7: Java applet of the two-dimensional Ising model – Play with the Java applet at

http://tinyurl.com/3fm6kcl

Set the system size to N = 100x100 spins and tune the temperature. Observe what happens when you cross the transition temperature T_c ~ 2.269 below which the model orders ferromagnetically.

last update: October 2011, v 1.3