

Lab 7 - Monte Carlo Methods

Thursday 16 October 2006 - Due: Thursday 23 October

In this lab, we will study Monte Carlo methods for estimating integrals. These techniques yield powerful methods for computing thermodynamic quantities in systems with very many degrees of freedom and, in that context, are generally referred to as Monte Carlo simulation which is a methodology widely applied throughout computational science.

1 A little about random numbers

1. Download the code `myrandom.py` demonstrated in lecture and paste it into an IDLE window. This program generates a sequence of pseudorandom numbers via the recurrence

$$r_{n+1} = (ar_n + b) \bmod m$$

This is the most common type of random number generator. If one chooses *very carefully* the constants a, b, m it can generate a stream of integers which passes most statistical tests of randomness. Most likely it is used in the python function `random()`. However, poor choices will generate a stream of numbers which are not at all random and indeed the sequence may cycle rather quickly through a fixed sequence of integers (note: to get floating point numbers in the range $(0, 1)$ one merely needs to divide by m .) Run the code. What is the repeat time of the sequence ?

2. Change to `a=106, b=1283` and `m=6075`. Do you see any repeat time now within the first 1000 calls of this generator ? If the numbers are carefully chosen the repeat time can approach m . Verify this by drawing more than 6075 random numbers.

2 Simple Monte Carlo

Consider the following quantity:

$$\langle x^2 \rangle = \frac{\int_{-b}^b dx x^2 e^{-\alpha x^2}}{\int_{-b}^b dx e^{-\alpha x^2}}$$

This resembles some of the integrals encountered in statistical mechanics restricted to the case of a single degree of freedom. The simplest Monte Carlo evaluation of this quantity is given by the formula

$$\langle x^2 \rangle \sim \frac{\frac{2b}{N} \sum_{i=1}^N x_i^2 e^{-\alpha x_i^2}}{\frac{2b}{N} \sum_{i=1}^N e^{-\alpha x_i^2}}$$

where the N sample points x_i are drawn at random in the interval $(-b, b)$

1. What line of python code will generate a number in the range $(-b, b)$? (Hint: you will need to use the `random()` function)

- Download the (partial) code `simplemc.py` from the webpage. Locate the comment line `## insert sample point here` and add the line of code you constructed in the previous question.
- The downloaded program contains a line of code to accumulate the numerator of the expression for $\langle x^2 \rangle$. Construct similar code to accumulate the denominator of $\langle x^2 \rangle$.
- Set $\alpha = 1.0$ and $b = 5.0$ and run the resulting code for $N = 100, 1000, 10000, 10^5, 10^6$. For each value of N do say $N_{\text{meas}} = 6$ separate runs and compute the average and standard deviation σ of your numbers. The standard deviation of some quantity f is given by

$$\sigma = \sqrt{\langle f^2 \rangle - \langle f \rangle^2}$$

where the angular brackets indicate performing an average over the measured values eg for quantity Q :

$$\langle Q \rangle = \frac{1}{N_{\text{meas}}} \sum_{i=1}^{N_{\text{meas}}} Q_i$$

The theoretical error in your Monte Carlo procedure is given by

$$\delta \langle f \rangle = \frac{\sigma}{\sqrt{N_{\text{meas}}}}$$

Plot your estimates for $\langle x^2 \rangle$ together with their estimated errors as a function of the logarithm to the base 10 of N . Hence show that the Monte Carlo estimates appear to converge to some fixed value and estimate that value together with its error using your largest N result.

- Repeat this for $b = 1000.0$. Is the error larger or smaller? You should see that the simple Monte Carlo algorithm becomes less efficient if the integrand is very small over substantial portions of the integration region. Essentially one wastes much of the calculation time sampling points at which the integrand contributes essentially nothing to the integral. This problem forces one to devise algorithms which use non-uniform sampling of integration points. Such *importance sampling* Monte Carlo algorithms can be very efficient if the sampling probability is chosen carefully. The most robust of these algorithms is that due to *Metropolis* and was discovered in the 1950's. It has since found wide application in large computational science simulations.

3 Metropolis Algorithm

It is generally more efficient to choose the sampling points not from a uniform distribution in the interval $(-b, b)$ but from a probability distribution which resembles the function to be integrated. In the example studied above it is natural to sample the integral by using points distributed according to the gaussian probability distribution $p(x) \sim e^{-\alpha x^2}$.

There exists a very robust way of generating a set of points distributed according to an arbitrary probability distribution termed the *Metropolis algorithm*. This algorithm starts

from some arbitrary point x_0 and generates a sequence of new sampling points x_i via some procedure such that the probability distribution of the set x_i approaches the target probability distribution after many steps of the algorithm. This procedure is not deterministic but employs random elements. It was described in detail in lecture.

1. Download the code `metro.py` from the webpage.
2. This simple program creates a set of N sample points using the Metropolis algorithm discussed in class. To get a good estimate of the error it is necessary, as for simple Monte Carlo, to make a series of such runs and measure the standard deviation of the resulting distribution of Monte Carlo estimates of the integral. In the previous section we did this error estimation by hand. It is more common to have the code do this explicitly by including a loop over the number of measurements. Add such a loop by inserting after the `## loop over measurements` line

```
for meas in range(0,MEASURE):
```

Remember to indent all the code relating to a single Monte Carlo run relative to this outer loop.

3. Now add the following lines after the line `mc/=N` (and flush with it)

```
mean+=mc
meansq+=mc*mc
```

4. Finally add a line of code to print out the mean and its Monte Carlo error. The latter is given by the formula you used before in terms of the standard deviation and the number of measurements.
5. Run the code for `MEASURE=2, 10, 50, 100`. Notice that the sampled points generated by the Monte Carlo are histogrammed over the run. Notice that they lie on a gaussian distribution as expected. Verify that the error falls as the inverse square root of the number of measurements. Can you guess the exact answer ?