Using the Pseudo-Random Number generator

Generating random numbers is a useful technique in many numerical applications in Physics. This is because many phenomena in physics are random, and algorithms that use random numbers have applications in scientific problems.

Most compilers come with a pseudo-random number generator. These generators use a numerical algorithm to produce a sequence of numbers that have many properties of truly random numbers. Although the sequence of pseudo-random numbers is not truly random, a good generator will produce numbers that give essentially the same results as truly random numbers. Most generators determine the pseudo-random number from previous ones via a formula. Once an initial number(s) (or seed(s)) is chosen, then the algorithm can generate the pseudo-random series.

A Simple Pseudo Random Number algorithm

A simple algorithm that will generate a sequence of integers between 0 and m is:

$$x_{n+1} = (ax_n + b)\ mod(m) \tag{1}$$

where a and b are constant integers. A sequence of integers x_i is produced by this algorithm. Since all the integers, x_i, generated are less than m, the sequence will eventually repeat. To have the period for repeating to be as large as possible, we want to chose m to be as large as possible. If m is very large, there is no guarantee that all integers less than m will be included in the sequence, nor is there a guarantee that the integers in the sequence will be uniformly distributed between 0 and m. However, for large m both these two properties are nearly satisfied and the algorithm works fairly well as a pseudo-random number generator.

For a 32-bit machine, a good choice of values are $a = 7^5$, $b = 0$, and $m = 2^{31} - 1$, which is a Mersenne prime number. Usually, one does not need to make up one's own pseudo-random number generator. Most C compilers have one built in.

Pseudo Random Numbers in C

There are various commands in C for generating random numbers. A common one is

random(32767)

This command returns a number with the properties of a random number with equal probability to lie between 0 and $32767 = 2^{16} - 1$. That is, a 16 bit random number

with uniform probability. To obtain a pseudo-random number between 0 and 1, the line of code:

r = random(32767)/32767.0;

does the trick. One can also include a line of code that sets the initial seed, or have the program pick a "random" seed. To generate a random number with uniform probability between a and b, the following code should work:

```
r = random(32767)/32767.0;
x = a + r*(b-a);
```

Generating a non-uniform probability distribution

Sometimes it is useful to generate random numbers that do not have a uniform distribution. This is fairly easy for the case of discrete outcomes. In this case, each discrete outcome i has a probability P_i of occuring, where the index i is countable (say from 1 to integer i_{max}). Note that P_i are unitless. For example suppose that you want to simulate an unfair coin: the coin is heads 40% of the time and tails 60% of the time. The following code does the task:

```
r = random(32767)/32767.0;
if (r ≤ 0.4) then heads;
if (r > 0.4) then tails;
```

In this example, one "throws" r with uniform probability distribution between zero and one. One then divides the interval between zero and one into the probability desired for the outcomes.

Often the outcome can be continuous. In this case the probabilities are represented as a probability density. Suppose the outcome is the continuous variable x, and the probability density $P(x)$. Then, $P(x)dx$ is the probability that the outcome lies between x and $x + dx$. Note that $P(x)$ has units of 1/(units of x), since $P(x)dx$ is unitless. Also note that normalization requires that $\int P(x)dx = 1$.

One can generate random numbers x with probabililty density $P(x)$ in a similar way as the discrete case. Below we describe the method, which will be proven in lecture:

1. Obtain a random number r with uniform probability between 0 and 1. That is, r = random(32767)/32767.0;

2. Solve the following integral for x:

$$r = \int_0^x P(u)du \tag{2}$$

x will be pseudo-random with a probability density $P(x)$.

We demonstrate this method with a simple example. Suppose you wanted to generate random numbers x between 0 and 1 with a probability density proportional to x: $P(x) = Cx$, where C is a constant. Normalization requires $\int_0^1 Cxdx = C/2 = 1$. So $C = 2$. Thus, the normalized probability density is $P(x) = 2x$. Now we need to relate x to r, which can be done by solving the integral above.

$$\begin{aligned} r &= \int_0^x 2udu \\ r &= x^2 \end{aligned}$$

or $x = \sqrt{r}$. A sample of code that would generate real numbers x with the probability density $P(x) = 2x$ might look like:

```
r = random(32767)/32767.0;
x = sqrt(r);
```

An important continuous non-uniform probability density is the Gaussian or Normal distribution with standard deviation σ:

$$P(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-x^2/(2\sigma^2)} \tag{3}$$

The following code, which will be explained in lecture, will generate random numbers x with a Gaussian probability distribution:

```
r1 = random(32767)/32767.0;
r2 = random(32767)/32767.0;
r = s * sqrt(-2*ln(1-r1)) ;
theta = 2*pi*r2;
x = r * sin(theta);
```

where s is the standard deviation.

Metropolis Algorithm (Monte Carlo)

The pseudo-random number generator is useful in evaluating multi-dimensional integrals, particularly those found in statistical mechanics. In statistical mechanics one often has to add up, or integrate, functions over configuration space, which is quite large. Sums of the form:

$$\sum_\mu f(\mu)w(\mu)$$

where $f(\mu)$ is some function of configuration "μ" of the system with probability $w(\mu)$ of occuring. The sum is over all possible configurations μ. For simplicity let's suppose that the number of possible configurations is finite and equal to N_{tot}. We can map each of the N_{tot} configurations to an integer. That is, μ will be an integer between 1 and N_{tot}.

One can evaluate this sum approximately by using a random walk method in "configuration space". The random walk will produce a sequence of configurations. For example, suppose $N_{tot} = 3$ and we label the thre possible configurations as $1, 2,$ or 3. That is μ equals 1, 2, or 3. A random walk sequence of configurations would be a random sequence of numbers that are 1, 2, or 3:

2, 3, 1, 1, 2, 1, 3, ...

We can label the j'th configuration in the sequence as μ_j. For example, in the sequence above, $\mu_1 = 2$, $\mu_2 = 3$, and $\mu_6 = 1$. Let N_{steps} be the number of steps in the random walk. If $N_{steps} >> N_{tot}$ the random walk will visit each configuration many times. Let $N(\mu)$ be the number of times we visit the configuration μ in the random walk. If we carry out the random walk such a way that $N(\mu) = N_{steps}w(\mu)$, then the random walk will be useful in evaluating the sum over configurations. This can be seen as follows. The sum over configurations can be written as:

$$\sum_\mu f(\mu)w(\mu) = \sum_\mu f(\mu)\frac{N(\mu)}{N_{steps}} \tag{4}$$

Note that the "trick" is to have the random walk visit the configuration μ a number of times $N(\mu)$ such that $N(\mu) = N_{steps}w(\mu)$. However, the sum $\sum_\mu f(\mu)N(\mu)/N_{steps}$ is equal to the sum $(\sum_j f(\mu_j))/N_{steps}$ since N_{steps} factors out of the sum. Thus, we have

$$\frac{\sum_j f(\mu_j)}{N_{steps}} = \sum_j f(\mu_j)\frac{N(\mu_j)}{N_{steps}} = \sum_\mu f(\mu)w(\mu) \tag{5}$$

which is what we wanted to calculate. This is a nice result. We just need to do the sum of $f(\mu_j)$ for a random walk through configuration space such that the sites μ are visited $w(\mu)N_{step}$ times. The trick is to find a way to generate an appropriate random walk. The algorithm of Metropolis et. al. is a clever way to produce such a random walk. Next, we state and justify the algorithm. An example program is located on the sample program web page, which allows you to check that the Metropolis algorithm has the desired characteristics.

Random Walk of Metropolis et. al.

The Metropolis et. al. algorithm (or Monte Carlo algorithm) is a method to obtain the $(j+1)$th element, μ_{j+1}, in the series from the jth element in the series, μ_j. The method is as follows:

1. Pick a trial configuration μ_t randomly, which is "close" to μ_j.
2. If $w(\mu_t) \geq w(\mu_j)$, then $\mu_{j+1} = \mu_t$.
3. If $w(\mu_t) < w(\mu_j)$, then $\mu_{j+1} = \mu_t$ with probability $r = w(\mu_t)/w(\mu_j)$, and $\mu_{j+1} = \mu_j$ with probability $(1-r)$.

One can understand why the random walk guided by these rules yields the correct probability as follows. Consider two configurations that could be in the random walk: μ and ν. Let's also suppose that $w(\nu) < w(\mu)$. Let $P(\mu \to \nu)$ represent the probability that ν is the next element in the random walk after μ if ν is chosen as the trial step. According to the rules above, $P(\mu \to \nu) = w(\nu)/w(\mu)$. Let $P(\nu \to \mu)$ represent the probabililty that μ is chosen after ν in the random walk if μ is the trial step. According to the rules of Metropolis, $P(\nu \to \mu) = 1$ since $w(\mu) < w(\nu)$. Therefore, we have:

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{w(\nu)}{w(\mu)} \tag{6}$$

We will next show that this condition will lead to $N(\nu)/N(\mu) = w(\nu)/w(\mu)$ for any two configurations μ and ν. Thus, the random walk will sample the configuration space with the number of visits per site proportional to the probability for the site.

Consider a finite size configuration space with N_{tot} sites. Each one is mapped to an integer μ as before. Suppose we have M_{tot} random walkers where $M_{tot} >> N_{tot}$, and that the walkers all change sites, "jump", at the same time. Let m_μ be the number of walkers at site μ at a certain time step. Then, the change in m_μ, Δm_μ, after the next step is given by:

$$\Delta m_\mu = \sum_{\nu \neq \mu} (+m_\nu P(\nu \to \mu) - m_\mu P(\mu \to \nu)) \tag{7}$$

where as before $P(\mu \to \nu)$ is the probability that a random walker will "jump" from configuration (or site) μ to configuration (or site) ν. The term on the left represents the number of walkers that jump to site μ, and the term on the right is the number of walkers that leave site μ and jump to site ν. After many many steps, N_{large} the system of random walkers will stabilize, and the average number of walkers at each site will not change. That is, eventually, $\Delta m_\mu \approx 0$ for each configuration μ. A solution to the steady state condition is for each term to vanish:

$$\begin{aligned} m_\nu P(\nu \to \mu) &= m_\mu P(\mu \to \nu) \\ \frac{m_\nu}{m_\mu} &= \frac{P(\mu \to \nu)}{P(\nu \to \mu)} \end{aligned}$$

after N_{large} steps. Suppose that there was one random walker that carried out $N_{large} M_{tot}$ steps. Then the number of times the walker visited site μ would be $N_{large} m_\mu$ times. Multiplying the numerator and denominator of the left side of the above equation by N_{large} yields:

$$\frac{N(\nu)}{N(\mu)} = \frac{P(\mu \to \nu)}{P(\nu \to \mu)} \tag{8}$$

The above argument demonstrates that for a random walk for which $N_{steps} >>> N_{tot}$ the ratio of the number of visits to site ν, $N(\nu)$ divided by the number of visits to site μ, $N(\mu)$ equals the inverse ratio of the probabilities for jumping between the two sites. Therefore, **if**

$$\frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{w(\nu)}{w(\mu)} \tag{9}$$

then

$$\frac{N(\nu)}{N(\mu)} = \frac{P(\mu \to \nu)}{P(\nu \to \mu)} = \frac{w(\nu)}{w(\mu)} \tag{10}$$

and the random walk will visit the configurations the correct number of times to give a correct estimate of the sum $\sum_\mu f(\mu) w(\mu)$. The Metropolis Algorithm satisfies this requirement.