## Random Numbers

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. A computer algorithm cannot produce true random numbers. Using a quantum system, or a system that is truly random, true random numbers can be produced. One can use radioactive decay to produce truly random numbers (see "Throwing Natures Dice", Ricardo Aguayo, Geoff Simms and P.B. Siegel, Am. J. Phys. 64, 752-758 (June 1996)) . However, it is difficult to produce these true random numbers quickly, and numbers that have the properties of true randomness can often be used. Therefore, it is useful to develop computer algorithms that generate numbers that have enough properties of true random numbers for scientific applications. Random numbers generated by a computer algorithm are called **pseudo-random** numbers.

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**

If you want to make your own pseudo-random numbers, 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. The series of numbers produced is fairly equally distributed between 1 and $m$. 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. I believe the defaut number in gcc is $2^{31} - 1 = 2147483647$. So in this case, we can use the following code:

```
m=pow(2,31)-1.0;
r=random()/m;
```

where $m$ is declared as double. I would recommend this defaut option for producing a pseudo-random number with uniform probability between zero and one.

One can generate a random number with uniform probability between $a$ and $b$ from a random number between 0 and 1. For example, the following code should work:

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

If $r$ is random with uniform probability between 0 and 1, then $x$ will have a uniform probability between $a$ and $b$.

## Generating a non-uniform probability distribution

### Discrete outcomes

Sometimes it is useful to generate random numbers that do not have a uniform distribution. This is fairly easy for the case of a finite number of discrete outcomes. For example, suppose there are $N$ possible outcomes. We can label each possibility with an integer $i$. Let the value of the $i$'th outcome be $z_i$. Let $P_i$ be the probability of obtaining $z_i$ as an outcome. The index $i$ is an integer from 1 to $N$. Note that the $P_i$ are unitless, and $\Sigma_{i=1}^{N} P_i = 1$.

One way to determine the outcome with the correct probability is as follows. Divide the interval $0 \rightarrow 1$ into N segments, where the $i$'th segment has length $P_i$. That is, the length of the $i$'th segment is the probability to have the outcome $z_i$. Then, "throw" a random number $r$ with uniform probability between 0 and 1. Whichever segment that $r$ is in, that is the outcome. In other words, if $r$ lies in the $k$'th segment, then the outcome is $z_k$. Since the probability to land in a particular segment is proportional to the length of the segment, the outcomes will have the correct probabilities.

Another way of expressing this idea, more formally, is the following. Define

$$A_n = \sum_{j=1}^{n} P_j \tag{2}$$

where $A_0 = 0$. $A_n$ is just the sum of the probabilities from $1 \rightarrow n$. Note that $A_N = 1$. Now, "throw" a random number $r$ that has uniform probability between 0 and 1. Find the value of $k$, such that $A_{k-1} < r < A_k$. Then the outcome is $z_k$.

We demonstrate the generation of discrete outcomes with a non-uniform distribution with two examples.

As a first 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 table below displays $z_i$, $P_i$, and $A_i$.

| $i$ | $z_i$ | $P_i$ | $A_i$ |
|---|---|---|---|
| 1 | heads | .4 | .4 |
| 2 | tails | .6 | 1 |

The following code will flip the coin with the desired probabilities:

r = random()/m;
if ($r \leq 0.4$) then heads;
if ($r > 0.4$) then tails;

As a second example, suppose you want to throw an unfair die with probabilities 0.2, 0.3, 0.1, 0.2, 0.1, 0.1, for the integers $1 \rightarrow 6$ respectively as shown in the table below:

| $i$ | $z_i$ | $P_i$ | $A_i$ |
|---|---|---|---|
| 1 | 1 | 0.2 | 0.2 |
| 2 | 2 | 0.3 | 0.5 |
| 3 | 3 | 0.1 | 0.6 |
| 4 | 4 | 0.2 | 0.8 |
| 5 | 5 | 0.1 | 0.9 |
| 6 | 6 | 0.1 | 1.0 |

In these examples, 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.

Next week we will discuss how to generate a non-uniform probability distribution when the outcomes are continuous quantities. The method is similar to the case of discrete outcomes that we just covered. We will also derive a method to produce random numbers that follow a Gaussian distribution, which you will use in assignment 4. Now let's cover some physics.

**Discrete outcomes**

Last week we discussed generating a non-uniform probability distribution for the case of finite discrete outcomes. An algorithm to carry out the non-uniform probability is to define an array $A$ as follows:

$$A_n = \sum_{j=1}^{n} P_j \tag{3}$$

where $A_0 = 0$. Note that $Z_N = 1$. $A_n$ is just the sum of the probabilities from $1 \rightarrow n$. Now, "throw" a random number $r$ that has uniform probability between 0 and 1. Find the value of $k$, such that $A_{k-1} < r < A_k$. Then the outcome is $z_k$. The following code should work after initiallizing the array $A[i]$:

```
i=0;
r=random()/m;
found=0;
while(found=0)
{
i=i+1;
if(A[i] > r) found=1;
if (i=N) found=1;
}
outcome=A[i];
```

**Continuous Outcomes**

In the realm of quantum mechanics the outcome of nearly every experiment is probabilistic. Discrete outcomes might be the final spin state of a system. In the discrete case, the probability for any particular outcome is a unitless number between 0 and 1. Many outcomes are however, continuous. Examples of continuous outcomes include: a particle's position, momentum, energy, cross section, to name a few.

Suppose the continuous outcome is the quantity $x$. Then one cannot describe the probability simply as the function $P(x)$. Since $x$ is continuous, there are an infinite number of possibilities no matter how close you get to $x$. One can only describe the randomness as the probability for the outcome $x$ to lie in a certain range. That is, the probability for the outcome to be between $x_1$ and $x_2$ $P(x_1 \rightarrow x_2)$ can be written as

$$P(x_1 \rightarrow x_2) = \int_{x_1}^{x_2} P(x)\, dx \tag{4}$$

The continuous function $P(x)$ can be interpreted as follows: the probability that the outcome lies between $x'$ and $x' + \Delta x$ is $P(x')\Delta x$ (in the limit as $\Delta x \rightarrow 0$). The probability function $P(x)$ has units of 1/length, and is refered to as a **probability density**. In three dimensions, the probability density will be a function of $x, y,$ *and* $z$. One property that $P(x)$ must satisfy is

$$\int P(x)dx = 1 \tag{5}$$

where the integral is over all possible $x$. In three dimensions this becomes:

$$\int\int\int P(\vec{r})dV = 1 \tag{6}$$

In quantum mechanics, the absolute square of a particle's state in coordinate space, $\Psi^*(x)\Psi(x)$ is a probability density. The function $\Psi(\vec{r})$ is a probability density amplitude. Likewise, the differential cross section is a probabililty density in $\theta$, with $f(\theta)$ being a probability density amplitude.

How does one produce a non-uniform probability for a continuous outcome from a random number generator that has a uniform probability distribution? We can use the same approach that we did for the discrete case. For the discrete case it was most useful to define $A_n$:

$$A_n = \sum_{j=1}^{n} P_j \tag{7}$$

Taking the limit to the continuous case, $P_j \to P(x')dx'$, and $A_n \to A(x)$. Suppose $x$ lies between $a$ and $b$, $(a \leq x \leq b)$. Then we have

$$A(x) = \int_a^x P(x')dx' \tag{8}$$

Note that $A(a) = 0$ and $A(b) = 1$ as before.

We can "throw" a random number with non-uniform probability density $P(x)$ as follows. First "throw" a random number $r$ between zero and one with uniform probability. Then solve the following equation

$$r = A(x) \tag{9}$$

for $x$. $x$ will be random with the probability density $P(x)$. This was the same method we used before in the case of discrete outcomes. After "throwing" $r$, the random outcome was the first value $n$ (for $A_n$) above $r$.

Another way of understanding this method for the case when $a \leq x \leq b$ is to start with the graph of $P(x)$, which goes from $a$ to $b$. Then divide the $a \to b$ segment on the x-axis up into $N$ equal pieces of length $\Delta x = (b-a)/N$. Now we have $N$ discrete outcomes, $x_j = a + j\delta x$, where $j$ is an integer and goes from one to $N$. The probability of each outcome equals the area under the curve in between $x$ and $x+\Delta x$: $P_j = P(x_j)\Delta x$. Now we define $A_n$ as before:

$$A_n = \sum_{j=1}^{n} P_j = \sum_{j=1}^{n} P(x_j)\Delta x \tag{10}$$

which is the equation we used in the discrete case. Taking the limit as $N \to \infty$ yields the continuous case.

Let's do an example. Suppose we want to throw a random number $x$ between zero and two that has a probability that is proportional to $x$. That is, $P(x) \propto x$, or $P(x) = Cx$. First we need to normalize $P(x)$, that is to find the constant $C$ such that $\int_0^2 P(x)dx = 1$.

$$\begin{aligned} \int_0^2 Cxdx &= 1 \\ C &= \frac{1}{2} \end{aligned}$$

Now we determine $A(x)$:

$$\begin{aligned} A(x) &= \int_0^x \frac{1}{2}x'dx' \\ A(x) &= \frac{x^2}{4} \end{aligned}$$

Now we "throw" a random number $r$ between zero and one with uniform probability. Then set $r = x^2/4$ and solve for $x$ in terms of $r$:

$$x = 2\sqrt{r} \tag{11}$$

If $r$ has a uniform probability between zero and one, then $x$ will have a (non-uniform) probability density of $x/2$.

The following computer code could be used to generate this non-uniform probability distribution for $x$:

```
r=random()/m;
x=2*sqrt(r);
```

**"Throwing" a Gaussian Probability distribution**

A Gaussian probability distribution in one dimension is the following function for $P(x)$:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \tag{12}$$

This probability density distribution is important in physics, because often experimental data will have random errors that follow this "normal" distribution. The average value of $x$ is $\mu$, with a standard deviation about this value of $\sigma$. Note, that here, $x$ ranges from $-\infty$ to $+\infty$.

To "throw" a Gaussian probability distribution in $x$, one would have to solve the following integral

$$A(x) = \frac{1}{\sigma\sqrt{2\pi}} \int_{\infty}^{x} e^{-(x'-\mu)^2/(2\sigma^2)} dx' \tag{13}$$

for $A(x)$. As far as I know, there is no analytic solution to this integral. However, there is a nice "trick" to produce a Gaussian probability. If one throws a two-dimensional Gaussian probability distribution, the integral can be solved analytically. I'll show you the result first, then show you how it is derived.

To throw a Gaussian distribution in $x$, with mean $\mu$ and standard deviation $\sigma$, do the following. Throw two random numbers, $r_1$ and $r_2$, each with a uniform probability distribution between zero and one. The random number $x$ equals $x = \mu + s\sqrt{-2\ ln(1-r_1)}\ sin(2\pi r_2)$. A simple code to carry this algorithm out is:

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

This algorithm is derived by tossing a two dimensional Gaussian probability in the x-y plane. If we "toss" a random point in the x-y plane that has a Gaussian probability density in $x$ and $y$ about the origin with standard deviation for both $x$ and $y$ as $\sigma$, the probability density is

$$P(x,y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \; \frac{1}{\sigma\sqrt{2\pi}} e^{-(y-\mu)^2/(2\sigma^2)} \tag{14}$$

The expression $P(x,y)$ is an area density, with units of 1/area. The complete expression for probability is

$$P(x,y)\; dxdy = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)} \; \frac{1}{\sigma\sqrt{2\pi}} e^{-(y-\mu)^2/(2\sigma^2)} dxdy \tag{15}$$

We can "toss" the point with the same probability density using the polar coordinates $r$ and $\theta$. The relationship in polar coordinates between $(x,y)$ and $r$ and $\theta$ is

$$\begin{aligned} x &= r\; cos(\theta) \\ y &= r\; sin(\theta) \\ & and \\ dx\; dy &= r dr\; d\theta \end{aligned}$$

Using these equations to transform to polar coordinates yields

$$\begin{aligned} P(x,y)\; dxdy &= \frac{1}{\sigma^2 2\pi} e^{-r^2/(2\sigma^2)} r dr\; d\theta \\ &= \frac{1}{\sigma^2} e^{-r^2/(2\sigma^2)} r dr\; \frac{d\theta}{2\pi} \end{aligned}$$

So if we "throw" $r$ with the probability density (in $r$) of $P(r) = (r/\sigma^2)e^{-r^2/(2\sigma^2)}$ and $\theta$ with a uniform probability density between 0 and $2\pi$, then the point $(r,\theta)$ in the x-y plane will have an x-coordinate that has a Gaussian probability distribution. The y-coordinate will also have a Gaussian probability distribution as well. However, you can not use both the $x$ and $y$ points in the same simulation. One does not get two indendent Gaussian probability distributions from the "$r-\theta$ toss".

To obtain $r$, first throw $r_1$ with uniform probability between zero and 1. Then $r$ is found by solving

$$\begin{aligned} r_1 &= \frac{1}{\sigma^2}\int_0^r e^{-r'^2/(2\sigma^2)} r\, dr \\ &= 1 - e^{-r^2/(2\sigma^2)} \\ &or \\ r &= \sigma\sqrt{-2\, ln(1-r_1)} \end{aligned}$$

$\theta$ is obtained by throwing $r_2$ with uniform probability between zero and 1. Then $\theta = 2\pi r_2$. Now that you see the derivation, you can understand why the code:

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

works. The mean $\mu$ is added to the Gaussian spread in the last line.

**Random Numbers in ROOT**

For those of you using ROOT, it is easy to "throw" a random number with a Gaussian distribution. There is a special function, rand.Gaus(mean,sigma). The following code should work:

```
TRandom3 rand(0); //set seed for random variable
x= rand.Gaus(mu, sigma);
```

Most likely the rand.Gauss function uses the method we described.