Lecture 7

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 $0 \to 1$ 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 $1 \to n$. 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.

Scattering amplitude and partial waves

In assignment 4 you are asked to simulate a scattering experiment. Your simulation will need to have some random error in the data. Before we discuss how to produce a Gaussian probability distribution for the random errors, let's go over the physics of scattering.

The differential cross section, $d\sigma/d\Omega$, can be written in terms of a scattering amplitude, $f(\theta, \phi)$. If the interaction is spherically symmetric, then there is no ϕ dependence for f. So

$$\frac{d\sigma}{d\Omega} = |f(\theta)|^2 \tag{3}$$

where $f(\theta)$ is a complex number with units of length. In our application, $f(\theta)$ will have units of fm^2. For non-relativistic energies, $f(\theta)$ can be determined from the Schroedinger equation with the appropriate scattering boundary conditions at $r = \infty$. If the interaction is spherically symmetric, i.e. $V(\vec{r} = V(r)$, then the Schroedinger equation can be separated into the different orbital angular momentum quantum numbers l. We derived this separation for the first assignment when we treated bound states. We obtained the set of equations:

$$-\frac{\hbar^2}{2m}(\frac{d^2u(r)}{dr^2} - \frac{l(l+1)}{r^2}u(r)) + V(r)u(r) = Eu(r) \tag{4}$$

with an equation for each value of l. The same separation holds for scattering problems. One will obtain a scattering amplitude for each value of orbital angular momentum l, which we label as f_l. The complete scattering solution will have a $Y_{lm}(\theta, \phi)$ added on for each l. For spherical symmetry, where there is no ϕ dependence, so the Y_{lm} reduce to Legendre polynomicals in $cos(\theta)$, $P_l(\theta)$. The scattering amplitude therefore becomes

$$f(\theta) = \sum_{l=0}^{\infty}(2l+1)f_l P_l(cos(\theta)) \tag{5}$$

The scattering amplitude $f(\theta)$ and the f_l are complex numbers, with a real and an imaginary part.

The sum over orbital angular momentum l in the expression for the scattering amplitude goes to infinity. The contribution from large l goes to zero, and one only needs to sum over a few values of l. The maximum value needed for l is roughly Rpc, where R is the size of the target and p is the momentum of the projectile. In our application we will only sum over the $l = 0$ and $l = 1$ **"partial waves"**.

One can express the partial wave amplitudes f_l in terms of "phase shifts". The phase shifts are determined from the solution of the Schroedinger equation for each l value. In terms of the **phase shifts**, δ_l, the elastic scattering amplitudes are

$$f_l = \frac{e^{i\delta_l} sin(\delta_l)}{k} \tag{6}$$

where $k = p/\hbar$.

You might wonder where the name phase shift comes from. The δ_l are the shift in phase from the free particle solutions of the Schroedinger equation. In the absence of the potential $V(r)$ ($V(r) = 0$), the solutions to the Schroedinger equation for orbital angular momentum l are the spherical Bessel functions, $j_l(kr)$. In fact, a plane wave traveling in the z-direction expressed in spherical coordinates is given by:

$$e^{ikz} = \sum_{l=0}^{\infty} (2l+1) i^l j_l(kr) P_l(cos(\theta)) \tag{7}$$

where θ is the angle with respect to the z-axis.

For large r, the spherical Bessel function $j_l(kr) \to sin(kr - l\pi/2)/(kr)$. The effect of a real potential $V(r)$ is to cause a phase shift in this large r limit: $sin(kr - l\pi/2 + \delta_l)/(kr)$. To solve for the phase shifts δ_l, one just iterates the Schroedinger equation to large r, like we did for the bound state assignment. However for the scattering calculation, the energy is greater than zero, and the solution oscillates. One can obtain the phase shifts by examining the large r solution to the discrete Schroedinger equation. We will not solve the Schroedinger equation for the δ_l in our assignment 4. I'll give you δ_0 and δ_1 for a particular energy, and you will generate simulated data.

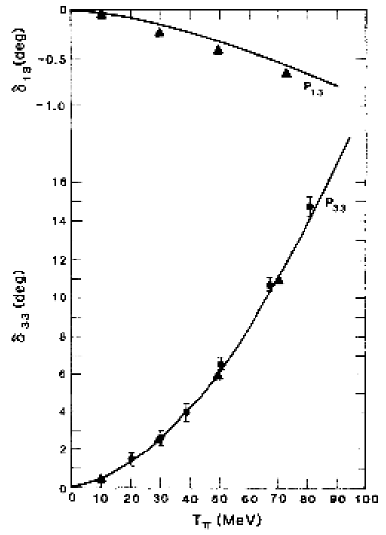

FIG. 3. Spin $\frac{1}{2}$ p-wave pion-nucleon phase shifts. The circles are from Ref. 15, the triangles from Ref. 16.

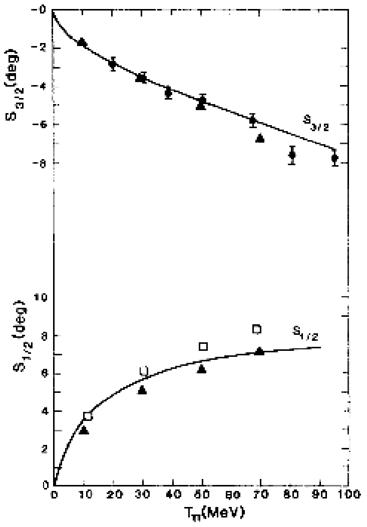

FIG. 5. s-wave pion-nucleon phase shifts. The circles are from Ref. 15, the triangles from Ref. 16, the boxes from Ref. 17.

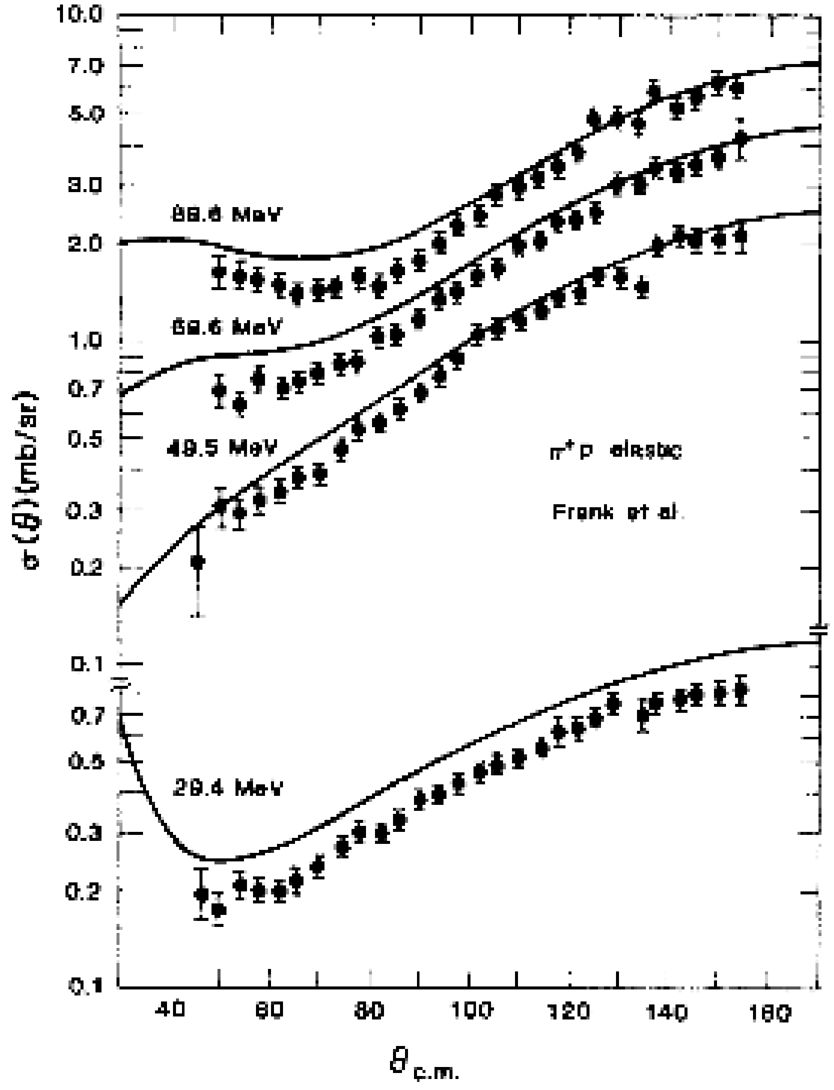

FIG. 7. Comparison with the data of Frank *et al.* (Ref. 19).