Daddy, Where do Random Numbers *Really* Come From?

In any of the MC procedures we have previously discussed we need many random numbers. A sequence of ***truly*** random numbers is, by its nature, unpredictable and therefore ***not*** reproducible.

A sequence of ***truly*** random numbers can ***only*** be generated by a random physical process, *e.g.* radioactive decay, thermal noise in a resistor, cosmic ray arrival times, ***etc***. If such a source of random numbers ***were*** used for MC calculations there would be no problem – the theory we have developed would work fine.

In practice, it is ***extremely*** difficult to construct a physical device that is fast enough to deliver random numbers from a random physical process as rapidly as modern computers demand them during a calculation. Several alternatives exist. For example, some years ago Argonne National Laboratory created a data file with 2.7×10^6 31-bit random numbers constructed from a U-235 α-particle source as follows:

Using a high-resolution ionization counter, it was found that in a $\Delta t = 20$ msec counting time interval, on average 24.315 α-particles were detected. When the count was *odd*, a 1-bit was recorded, and when the count was *even*, a 0-bit was recorded. Then the bits were strung together as 31-bit sequences of (unsigned) integers.

Bias removal in this data set was accomplished as follows: Suppose one has a sequence of 0's and 1's which ***is*** random, but the probabilities P(0) and P(1) are ***not*** both ***precisely*** equal to 1/2.

Divide the sequence into pairs, *e.g.* 01 10 10 10 11 01 10 01 00 10 ...

Ignore 00 and 11 pairs, so the above will become: 01 10 10 10 01 10 01 10 ...

For the remaining pairs, drop the ***first*** bit.
This will yield: 1 0 0 0 1 0 1 0 ...

Now the probability of finding a 0 or 1 in the final sequence, $P'(0)$ and $P'(1)$ are:

$P'(0) = P(1)\,P(0)$ (since it came from a (10) pair)

$P'(1) = P(0)\,P(1)$ (since it came from a (01) pair)

Thus, $P'(0) = P'(1)$, *i.e.* 0's and 1's ***are*** indeed equally likely in the ***final*** sequence.

Unfortunately this procedure is very wasteful, since about half of the bits are thrown out right away, and half of the remaining ones are not used.

The common choice in modern applications is a source of "**Quasi-random**" (or "**pseudo-random**") numbers. This is what is produced by random number generators on a computer. The numbers are generated according to a strict mathematical procedure leading to a sequence which is predictable (in advance) and certainly reproducible. Thus "random numbers" on a computer are *not at all random* in the mathematical sense .... however, they are ***supposed*** to be indistinguishable from a sequence generated truly randomly. That is, if one didn't know that the numbers were generated by a strict mathematical procedure but one was just given the sequence, then one would be unable to tell that a formula was used rather than a physical process.

Unfortunately, no one has yet learned how to generate quasi-random numbers like this.

One **can** tell the difference! However, this does not prevent people from using sequences of such numbers as if they were truly random (or even truly quasi-random) and ignoring the fact that, theoretically, they'e on shaky ground…

For a long time the most widely used random # generator was the "**multiplicative congruential**" :

$$\boxed{r_i = \left(ar_{i-1} + b\right) \bmod m}$$

where a and b are constants and $\bmod m$ means take the remainder after division by m. *i.e.* the i^{th} random # is the ***remainder*** when $\{a(i-1)^{\text{th}}$ random # $+ b\}$ is divided by m.

Since taking the operation of taking the remainder commutes with the division operation, we can alternatively work with a pseudo-random integer sequence, and then divide the sequence of integers by m. For example, if $a = 13$, $b = 0$, $m = 31$ and $r_0 = 1$, then this pseudo-random ***integer*** sequence begins with: 1, 13, 14, 27, 10, 6, 16, 22, 7, 29, 5, 3, … the next random # is $r_i = \left(13 \cdot 3 + 0\right) \bmod 31 = 39 - 31 = 8$.

The first 30 terms in this sequence are a permutation of the *integers* 1–30, but then the random # sequence ***repeats*** itself, *i.e.* it has ***period*** $m - 1$.

If the pseudo-random integer sequence such as the one above is then scaled by dividing by m, the result are floating-point #'s that are uniformly distributed on the interval [0,1]. In our above example: 0.0323, 0.4194, 0.4516, 0.8710, 0.3226, 0.1935, 0.5161, …

Obviously, one doesn't want a sequence of random #'s with a short period, so one chooses m to be a very large integer, along with choices for a and b. For example, in the 1960's, Scientific Subroutine Package (SSP) on IBM mainframe computers had a multiplicative congruential random # subroutine RND (or RANDU) for use with 32-bit computation, with $a = 2^{16} + 3 = 65539$, $b = 0$ and $m = 2^{31} = 2147483648$. Because of the choice $a = 2^{16} + 3$, note that the multiplication operation ar_{i-1} can be carried out by a shift operation + addition operation, which was motivated by speed considerations on such computers of that era. It can be shown that for this choice of IBM's random # parameters that:

$$\left(r_{i+1} = 6r_i - 9r_{i-1}\right) \bmod 2^{31} \quad \text{for all } i$$

i.e. there were extremely high correlations between 3 sequential random #'s in IBM's code!

Over the years, collectively we have learned that the success of a random # generator is not related to its complexity. In fact, an arbitrary invention of great complexity is:

a.) probably very hard to analyze as far as its properties are concerned, and
b.) likely to be a poor generator, giving very un-random like sequences.

During the 1960's, many random # generators were tried and tested.

By the mid-1970's, only the multiplicative congruential type were seriously used.

Normally m is chosen as 2^t, where t is the number of bits used to represent an integer on the machine. Since a and r_{i-1} are t-bit integers, their product has $2t$ bits. After b is added (often $b = 0$) the lowest t bits are kept as r_i. Finally, r_i is divided by m to give a floating point number between 0.0 and 1.0 .

One property of most random # generators, including the multiplicative congruential ones, is their period. Obviously, once any specific number occurs a second time, the sequence following it is the same thereafter. For multiplicative congruential random # generators, the period is dependent on a, b, and r_0, the initial, or "seed" random number which must be chosen before the random # sequence is generated.

The maximum period can be shown to be of length (m/4). For typical 32-bit computers today, $t = 31$ (the remaining bit is used for a sign) and $2^t = 2\ 147\ 483\ 648$, which limits the maximum length of a random # sequence to about 500 million.

<u>Testing Random # Generators</u>:

In the past, it was dangerous to depend on the random # generators supplied with the subroutine/ function libraries of some computer systems. For example, as mentioned previously RANDU (distributed by IBM with the 360), was found (right away) to be poor, but that was already too late! Even 10-20 years later (*i.e.* in the 1970's–80's) people were *<u>still</u>* using RANDU! *Eeeeek*!!!

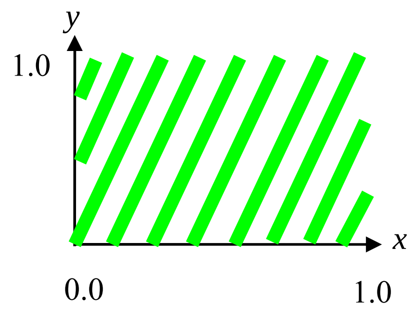
In the "dark" ages (1960's) there was no deep understanding of quasi-random # generators and so the tests were not very specific. Typically, just "apparent randomness" was tested. For example, consider the sample mean of a sequence of N random #'s: $\bar{r} = \frac{1}{N}\sum_{i=1}^{N} r_i$

If the r_i are a set of N random numbers drawn from $U(0,1)$ then we know that $\hat{r} = 1/2 = 0.5$ and that $\sigma_{\bar{r}} = \left\{1/\sqrt{12}\right\}/\sqrt{N}$.

So we check the generator to see if $\bar{r}$ is within 2σ of 0.5, which it should be 95% of the time. If the random # generator were to fail this test, we would say that "it fails the test at the 5% level". Note – this actually proves nothing, since for ***<u>true</u>*** random #'s, we expect them to fail 5% of the time! If the random # generator passes this test, we might go on to look at $\sigma_{\bar{r}}$, which we find by generating a long sequence of random numbers and dividing it up into groups of size N.

In reality we use batteries of more complicated tests. Then we <u>assume</u> that, having passed all of these tests, the generator will also pass the next test. The "next" test is the requirement that it be "random enough" to work properly in <u>your</u> problem. It is not known, in general, why it should pass this test, but it isn't known, either, why it should not.

A very different kind of test was discovered in 1962 by a high-energy physicist (Joe Lach) who was trying to understand why the random # generator that came with IBM's 709 gave larger fluctuations than expected. He displayed the random #'s on a CRT. (In those days, this was not a common procedure!) When he divided his random # sequence into pairs and plotted them as (x,y) coordinates of points, he saw nothing strange, just a uniform density of points. Then he divided it into triplets, considering each as the (x,y,z) of a point in 3-D space. When he plotted the (x,y) coordinates for all triplets for which $z < 0.1$ he saw the points falling into a set of slanted bands with ***<u>no</u>*** (x,y) values in between, as shown in the figure below:



This phenomenon was later shown by George Marsaglia to be a "defect" (*i.e.* a feature) of ***all*** multiplicative congruential random # generators. In *Random Numbers Fall Mainly in the Planes*, (PNAS, Vol 61, 1968) he showed that if successive *d*-tuples from such a random # generator are taken as coordinates of points in *d*-dimensional space, all of the points will lie on a certain finite number of parallel hyperplanes. (The original discovery saw 3-D planes projected onto 2-D. They are "tilted" and thus appear as bands in 2-D.)

The ***maximum*** number of hyperplanes depends on d and the number of bits used for integer representation on the particular computer being used:

The maximum number of hyperplanes $= \left(d!2^t\right)^{1/d}$. This gives:

t	$d = 3$	$d = 4$	$d = 6$	$d = 10$
16	73	35	19	13
32	2953	566	120	41
60	1905389	72527	3065	289
64	4801280	145055	4866	382

Moral: ***Never*** trust a random # generator on a 16 bit machine, unless it uses double-precision!!!

Even 32 bits ***may*** be dangerous, *e*.g. for carrying out a 10-dimensional integral.

Warning: Maximum number is just that. For an arbitrary choice of the parameters a and b in $r_i = \left(ar_{i-1} + b\right) \bmod m$ it can be smaller.

As a result of Marsaglia's work, and other efforts at the same time (early 70's) we now understand multiplicative congruential random # generators very well.

Still, it is not safe to assume that the people who supply them with computers understand them. One should only trust those with "pedigrees", *e.g.* those from CERN, SLAC, ANL, ORNL, *etc*. Often these have been designed to give "better" random #'s. For example, it has been shown that by using a random # generator such as $r_i = \left(ar_{i-1} + br_{i-2}\right) \bmod m$ where a and b are chosen ***carefully*** one can increase the number of hyperplanes by a factor of $2^{t/d}$.

Since the 70's, considerable progress has been made on developing several new types of random # generators that don't have the above-mentioned problems that multiplicative congruential random # generators suffer from. One such random # generator is RANLUX, which has a solid theoretical basis in chaos theory, based on the work of M. Lüscher, which in turn is based on work by G. Marsaglia and A. Zaman, regarding so-called lagged Fibonacci sequence generators. These generators and other modern random # generators are efficient and have very long repeat periods of up to $\sim 10^{43}$ and pass extensive "DIEHARD" or TESTU01 batteries of tests, for which the most commonly available multiplicative congruential random # generators fail, with repeat periods of less than $2^{32} \sim 4.3\times10^{9}$.

The interested reader is referred to the following papers:

1.) "Uniform Random Number Generators: A Review", Pierre L'Ecuyer, Proc. 1997 Winter Simulation Conference, IEEE Press, Dec. 1997, 127-134.
2.) "A review of pseudorandom number generators", F. James, Comp. Phys. Comm. **60**, 329-344 (1990).
3.) "RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher", F. James, Comp. Phys. Comm. **79**, 111 (1994).
4.) "A portable high-quality random number generator for lattice field theory simulations", M. Lüscher, Comp. Phys. Comm. **79** 100 (1994).
5.) "Maximally Equidistributed Combined Tausworthe Generators", Pierre L'Ecuyer, Mathematics of Computation **65**, 213 (1996).
6.) "Tables of Maximally Equidistributed Combined LFSR Generators", Pierre L'Ecuyer, Mathematics of Computation **65**, 225 (1999).
7.) "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", M. Matsumoto and T. Nishimura, ACM Transactions on Modeling and Computer Simulations, Vol. 8, No. 1, January 1998, 3-30.
8.) "Some difficult-to-pass tests of randomness", G. Marsaglia and W.W. Tsang, Journal of Statistical Software, Volume 7, 2002, Issue 3.
9.) "Some portable very-long-period random number generators", G. Marsaglila and A. Zaman, Computers in Physics, Vol. 8, No. 1, Jan/Feb. 1994.
10.) "A New Class of Random Number Generators", G. Marsaglila and A. Zaman, The Annals of Applied Probability 1991, Vol. 1, No. 3, 462-480.
11.) W. H. Press, *et al.*, Numerical Recipes, 3rd Ed., Cambridge University Press, New York, 2007.
12.) D.E. Knuth, The Art of Computer Programming, Vol. II, 3rd Ed., Addison-Wesley, Reading, Mass. 1998.

The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness is available on the web at the following URL: http://www.stat.fsu.edu/pub/diehard/

Info on the TESTU01 test suite is available on the web at the following URL: http://www.iro.umontreal.ca/~simardr/testu01/tu01.html

Info on Dieharder, another random # generator test suite is available at the following URL: http://www.phy.duke.edu/~rgb/General/dieharder.php

High-quality random # generators are ***essential*** in today's world of ever-increasing compute power for use in ***ciphers & encryption*** – *i.e.* ***secure communications*** and ***financial transactions*** *e.g.* on the internet and elsewhere…

Quasi-Monte Carlo:

If we in use sequences of quasi/pseudo-random numbers for Monte Carlo, then we are factually doing "Quasi-MC", but in practice we still call it "Monte Carlo". A theoretical basis exists that justifies the use of Quasi-MC, based on the work of Weyl. The bottom line is that it works and typically shares the properties of a ***true*** MC, such as $1/\sqrt{N}$ convergence.

Non-Uniform Random Numbers:

We frequently need to generate random #'s with P.D.F.'s other than $U(0,1)$.We have already seen how to generate $U(a,b)$, as $x_i = a + (b-a)r_i$, however often we need non-uniform random #'s. A long time ago (in early lectures), we saw a trick for generating $N(0,1)$ from r_i which are $U(0,1)$: $\sum_{i=1}^{12} r_i - 6$ is approximately Gaussian, with expectation value zero and variance one. However, it has no tails beyond $\pm 6\sigma$ (*i.e.* *here*, ± 6).

Gaussian Random Number Generation:

If we define (the Box-Muller transformation):

$$z_1 = \sqrt{-2\ln u_1}\cos(2\pi u_2)$$
$$z_2 = \sqrt{-2\ln u_1}\sin(2\pi u_2)$$

where (u_1, u_2) are ***independent*** and drawn from $U(0,1)$. Then z_1 and z_2 are ***independent*** and each is distributed as $N(0,1)$, *i.e.* $g(z_1,z_2) = \frac{1}{\sqrt{2\pi}} e^{-z_1^2/2} \frac{1}{\sqrt{2\pi}} e^{-z_2^2/2}$

Random Triangular P.D.F.

We also saw earlier that if u and v are independent and drawn from $U(0,1)$ then $u + v$ has a *triangular* P.D.F.

Random Square-Root P.D.F.

We can also show that if we generate (u,v) pairs but keep only the ***larger*** one (call it *x*), then x is distributed as $\sqrt{x}$.

A General Method for Non-Uniform Random Numbers:

Suppose we are given some P.D.F. $g(y)$ and we wish to generate a sequence of random #'s distributed according to $g(y)$. Suppose we have available a sequence of random #'s *x*, distributed as $U(0,1)$. Then:

$$g(y)dy = f(x)dx = 1dx$$

It turns out to be useful here to use the Cumulative Distribution Function $G(y)$ which is related to the P.D.F. by:

$$g(y) = dG(y)/dy$$

Recall that $G(y)$ is the probability that the random variable will be have a value $\leq y$, and that $0 \leq G(y) \leq 1$. In general, we can find $G(y)$ from $g(y)$ by integration: $G(y) = \int_{-\infty}^{y} g(y')\,dy'$

n.b. here "$-\infty$" stands for the smallest value of y allowed by the P.D.F., which might in fact be $-\infty$.

Let us rewrite the integral in terms of x. The smallest allowed value of x is 0, and let x correspond to the chosen y:

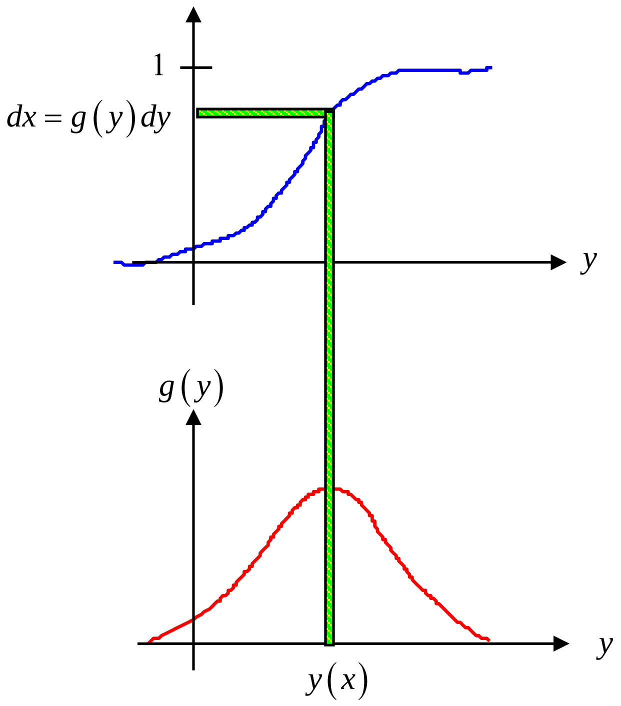
$$G(y) = \int_{-\infty}^{y} g(y')\,dy' = \int_{0}^{x} f(x')\,dx'$$

But x is still some random number distributed according to $U(0,1)$, call it "*r*". Thus:

$$G(y) = \int_{-\infty}^{y} g(y')\,dy' = r$$

If we can ***invert*** this, *i.e.* solve for y as a function of r, then we're done, that is: $y = G^{-1}(r)$ will have the desired P.D.F.

$$x = G(y) = \int_{-\infty}^{y} g(y')\,dy'$$



Here, *x* lies between 0 and 1 in an interval *dx*. (Both *dx*'s are the same.) We project onto the function *G*(*y*).

The corresponding *y* is more likely to be in a narrow interval *dy* in a region where *G*(*y*) is rapidly varying (*i.e.* near the maximum of *g*(*y*)) rather than where *g*(*y*) is small, where *G*(*y*) is slowly varying.

Random Exponential Distribution:

Let us distribute random numbers exponentially:

$$f(x) = \begin{cases} ce^{-\alpha x} & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Here: $$F(x) = \int_{-\infty}^{x} f(x')\,dx' = \int_0^x ce^{-\alpha x'}dx' = \frac{c}{\alpha}\left(1-e^{-\alpha x}\right)$$

In order to normalize this function, we require $F(\infty)=1$, or $c=\alpha$.

$$\therefore \quad f(x) = \begin{cases} \alpha e^{-\alpha x} & x \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad F(x) = 1-e^{-\alpha x}$$

In order to generate random numbers distributed in this way, just pick an r_i from $U(0,1)$, then set $r_i = F(x_i)$ and solve for x_i:

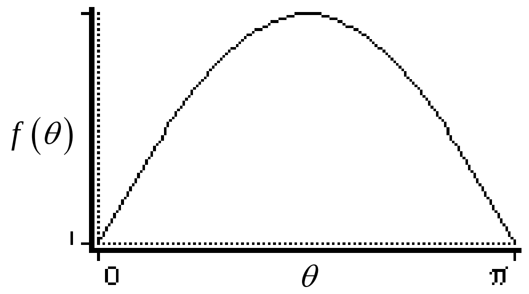
$$r_i = 1-e^{-\alpha x_i} \quad \text{or:} \quad 1-r_i = e^{-\alpha x_i} \quad \text{or:} \quad x_i = -\left(\ln(1-r_i)\right)/\alpha$$

Note that since r_i is drawn from $U(0,1)$, then so is $1-r_i$, and hence with no loss of generality we can instead use: $x_i = -(\ln r_i)/\alpha$ (and save a bit of computer time).

Exponential distributions are often encountered as lifetimes of unstable states ($e^{-t/\tau}$, so that $\alpha = 1/\tau$) or in the absorption of radiation by matter ($e^{-x/\lambda}$ where λ = mean free path), *etc.*

Uniform cosθ Distribution:

Another example: $f(\theta) = \begin{cases} c\sin\theta & \text{for } 0 \leq \theta \leq \pi \\ 0 & \text{elsewhere} \end{cases}$



Now $F(\theta) = \int_{-\infty}^{\theta} f(\theta')\,d\theta' = \int_0^{\theta} c\sin\theta' d\theta' = c(1-\cos\theta)$

Demanding $F(\pi)=1$ gives $c = 1/2$. $\quad \therefore \quad f(\theta) = \tfrac{1}{2}\sin\theta$ and: $F(\theta) = \tfrac{1}{2}(1-\cos\theta)$

So if r_i is drawn from $U(0,1)$, then we set $r_i = \tfrac{1}{2}(1-\cos\theta_i)$.

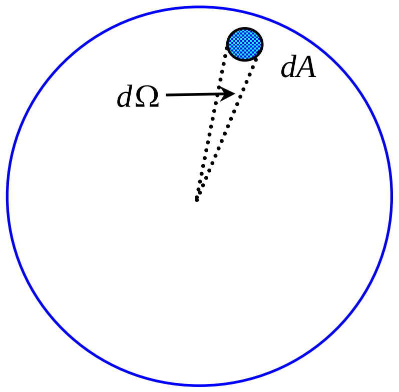
This gives: $\cos\theta_i = 1-2r_i$ or: $\theta_i = \cos^{-1}(1-2r_i)$. Again, with no loss of generality we can set $\theta_i = \cos^{-1} s_i$ where s_i is drawn from $U(-1,1)$. This also follows directly from the following:

We can approach this same problem slightly differently: $f(\theta)d\theta = \tfrac{1}{2}\sin\theta d\theta = -\tfrac{1}{2}d\cos\theta$

This means that a "$\sin\theta$" distribution in the variable "θ" corresponds to a **Uniform** distribution in the variable "$\cos\theta$" over the same region. So if we draw $\cos\theta$ from $U(-1,1)$ we will get the same angle distribution as throwing θ according to $\sin\theta$.

Random Isotropic Direction in 3-D:

We can apply the above idea to the problem of generating a ***random unit vector***, *i.e.* a vector whose magnitude is 1 but which can point in any direction on the unit sphere.



Then the number of resulting vectors pointing into any dA should be proportional to dA.
But $dA = r^2 d\Omega = 1 d\Omega$ on the unit sphere, and $d\Omega = \sin\theta d\theta d\varphi = |d\cos\theta| d\varphi$.

However, θ and φ are ***independent*** random variables and hence we can "throw" them independently by choosing random $\cos\theta$ drawn from $U(-1,1)$ and φ drawn from $U(0,2\pi)$. Then this will give the desired distribution, uniform in $d\Omega$.

i.e. explicitly, choose: $\begin{cases} \cos\theta_i = 1-2r_i \\ \varphi_i = 2\pi s_i \end{cases}$ where r_i and s_i are drawn from $U(0,1)$.

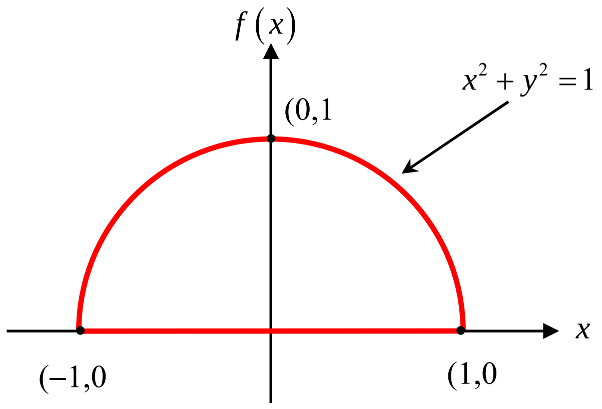
Then the unit vector $\hat{v} = (\sin\theta_i\cos\varphi_i, \sin\theta_i\sin\varphi_i, \cos\theta_i)$ will be distributed randomly over the unit sphere.

This method won't ***always*** work for a ***general*** $g(y)$ because:

- It may not be possible to integrate and get $G(y) = \int_{-\infty}^{y} g(y')dy'$ in a simple form (*e.g.* in terms of elementary functions)
- It may not be possible to invert and solve $r_i = F(x_i)$ for $x_i = F^{-1}(r_i)$

<u>Random Uniform Distribution on a 2-D Disk</u>:

As an another example, let us randomly, uniformly populate the interior of a semicircle.



Specifically, we want the number of x's generated in the interval $x \to x + dx$ to be proportional to $\sqrt{1-x^2}$. Thus, the P.D.F. we want to generate according to this will be:

$$f(x) = \begin{cases} \sqrt{1-x^2} & -1 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$$

The corresponding Cumulative Distribution Function is:

$$F(x) = \int_{-\infty}^{x} f(x')\,dx' = \int_{-1}^{x} c\sqrt{1-x^2}\,dx' = \frac{c}{2}\left[x'\sqrt{1-x'^2} + \sin^{-1} x'\right]_{-1}^{x}$$

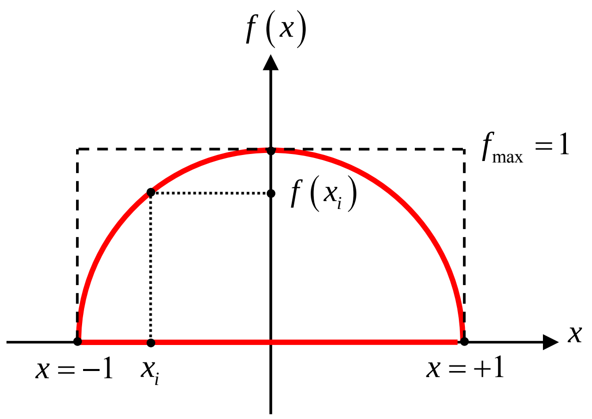
If we set $F(x) = r$, a random #, we will ***<u>unable</u>*** to invert this and solve for *x* as a function of *r*.

In such cases we can use the "**Von Neumann MC Acceptance-Rejection Method**", a kind of "hit-or-miss" MC technique. Here, for example, we:

(1) Pick a random # x_i drawn from $U(-1,1)$ and a random # y_i drawn from $U(0,1)$.

(2) We then calculate $f(x_i) = \sqrt{1-x_i^2}$.

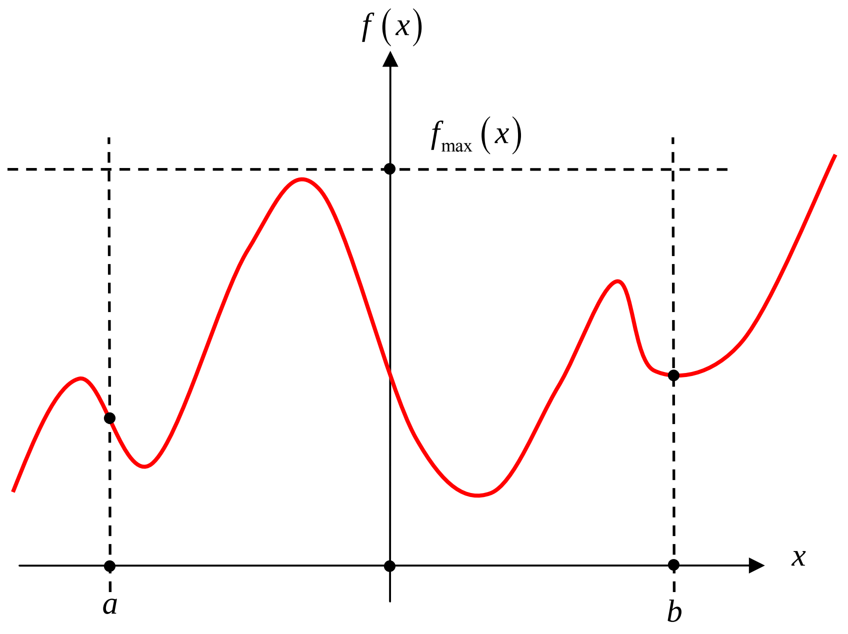
(3) If $y_i < f(x_i) = \sqrt{1-x_i^2}$ we ***<u>keep</u>*** x_i; otherwise we ***<u>reject</u>*** x_i and go to step (1) and repeat…



Step (1) uniformly populates the ***outer*** dotted rectangle. By ***rejecting*** x_i's for which the corresponding y_i is ***outside*** the ***red*** semi-circle, we end up with "hits", *i.e.* x_i's such that the number of ***kept*** x_i's between $x \to x+dx$ is proportional to $f(x)dx$, being large where $f(x)$ is large and small where $f(x)$ is small, i.*e*. we randomly populate the ***interior*** of the semi-circle by ***rejecting*** points that fall ***outside*** of the semi-circle.

We now generalize the MC "hit-or-miss" (*aka* the "acceptance-rejection") method:

Suppose we have a general/arbitrary function $f(x)$ in the range $a \le x \le b$, and wish to generate random #'s x_i as if the function were a P.D.F. between $a \le x \le b$, as shown in the figure below:



Find/determine the value $f_{\max}$ such that $f(x) \le f_{\max}$ in the range $a \le x \le b$. Then:

(1) Pick a random # x_i drawn from $U(a,b)$ and a random # y_i drawn from $U(0,1)$.
(2) We then evaluate $f(x_i)$.
(3) If $y_i < f(x_i)$ we ***keep*** x_i; otherwise we ***reject*** x_i and go to step (1) and repeat…

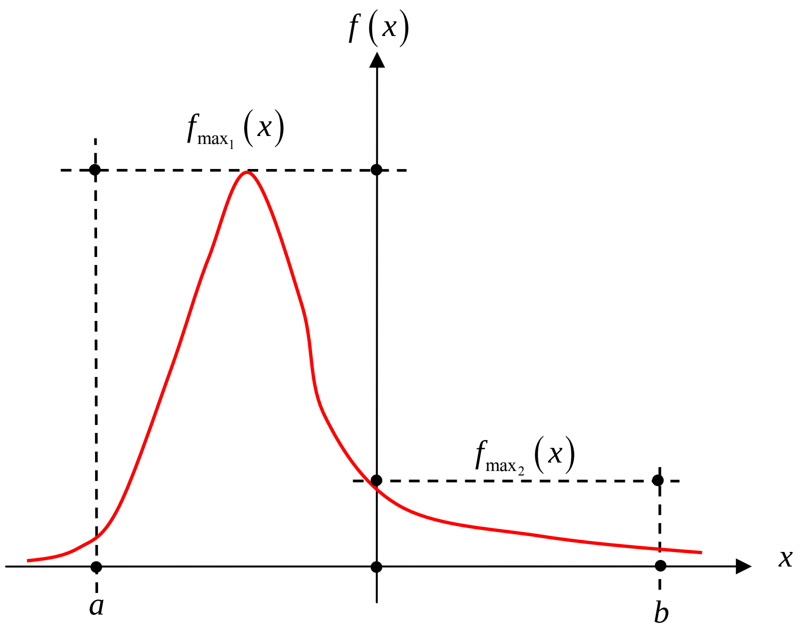
Note that this technique also gives $\int_a^b f(x)\,dx$ if we keep track of the # of ***tries*** and the # of ***successes***, since:

$$\frac{\#\text{ of successes}}{\#\text{ of tries}} = \frac{\text{area under } f(x)}{\text{area under rectangle}} = \frac{\int_a^b f(x)\,dx}{(b-a)\,f_{\max}}$$

It is also straightforward to generalize this method to the case of a multi-dimensional function.

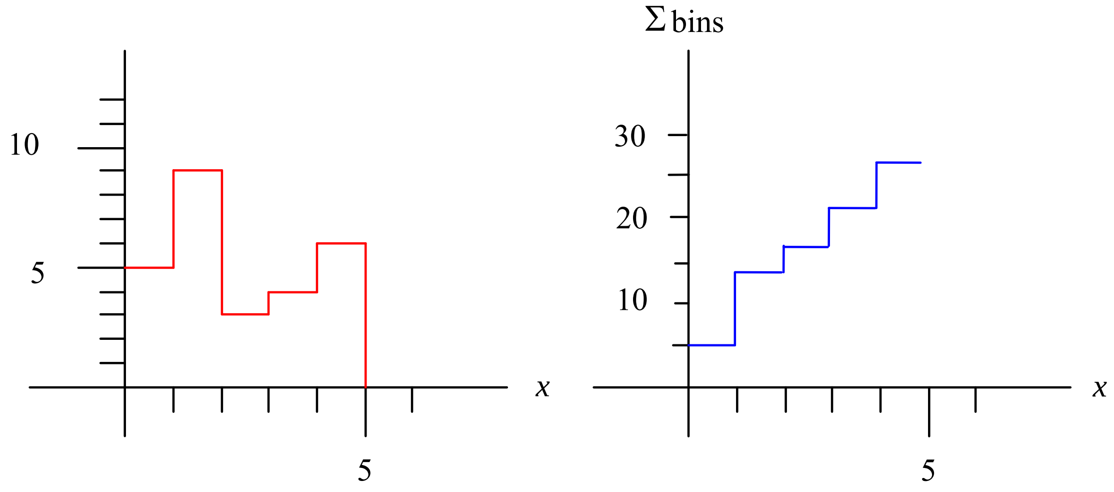
The MC "acceptance-rejection" method will be most efficient (*i.e.* fewest # of tries) when $f_{\max}$ is as small as possible, however this smallest value may not be simple to find in the more general case where the function f is a very complicated multi-dimensional function.

Depending on the details of the shape of $f(x)$, one can improve the efficiency of the above acceptance-rejection algorithm by using an (elementary) ***importance-sampling technique***. In the figure shown below, the $f(x)$ is such that we can use ***two*** rectangular boxes to improve the efficiency/speed of this algorithm:



Random Numbers from Histogrammed Distributions:

We can also generate random #'s that follow distributions that cannot be described analytically by simple continuous functions. For example, suppose we wish to generate random x_i's according to a ***histogram***:



The ***given*** histogram is on the left, its associated Cumulative histogram on the right.

Since the sum of the contents of ***all*** bins is 27, we throw a random number r according to $U(0,27)$ and use a table:

If:	Keep:
$0 \le r \le 5$	$x = 0$
$5 < r \le 14$	$x = 1$
$14 < r \le 17$	$x = 2$
$17 < r \le 21$	$x = 3$
$21 < r \le 27$	$x = 4$

We can use an analogous procedure if we are given a set of ***discrete*** probabilities. For example, suppose we know that:

$$P(0.0 \le x \le 0.3) = P_1$$
$$P(0.3 \le x \le 0.9) = P_2 \quad \text{where:} \quad P_1 + P_2 + P_3 = 1$$
$$P(0.9 \le x \le 1.0) = P_3$$

Then we obtain random # p drawn from $U(0,1)$.

If $0 < p \le P_1$ we select the x-interval $0.0 \le x \le 0.3$.
If $P_1 < p \le P_1 + P_2$ we select the x-interval $0.3 \le x \le 0.9$.
Otherwise we select the x-interval $0.9 \le x \le 1.0$.

Then we must decide separately (we are not given any information here) how to pick a particular value of x in one of these intervals. We could *e.g.* pick $x = 0.15$ each time we landed in the first region, or we could pick x from $U(0.0, 0.15)$, or ...

The same procedure can be used when simulating a physical process. At some instant of time or at a point in 3-D space, a choice must be made between different outcomes with different probabilities.

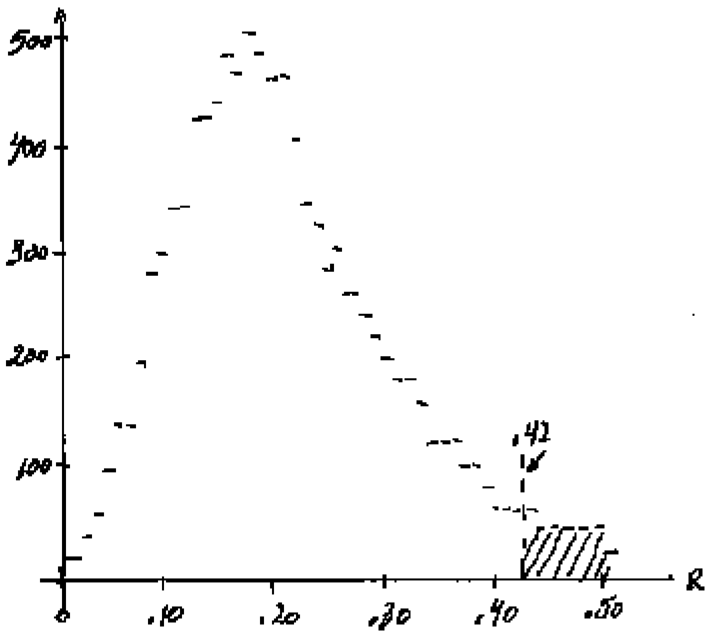
For example, a neutron, having arrived at some point in the matter it is traversing, can at that point undergo an elastic scattering, be absorbed by a nucleus, *etc*., each with different probabilities (which depend on the material and the energy of the neutron).

MC techniques are commonly used to do integrals, simulate physical processes to answer physics questions, *etc*. In high energy physics they are frequently used to simulate data which resembles as closely as possible the expected signals from real detectors. These are then used as input to the programs that "reduce" the signals (*e.g.* obtain digitized times, pulse heights, *etc*.) to meaningful quantities, and (perhaps) fit them to theoretical models which extract basic parameters. Since the MC input is completely understood, the data analysis programs can be checked to see if they work as they are intended to.

A less common MC application is to the **Propagation of Large Errors**. Here "large" means that we cannot use standard approximations for combining errors and we must calculate, exactly, the P.D.F. of the result of some combination of measurements, which may be very difficult to do analytically.

An interesting example is provided by a tritium β -decay experiment (no longer taken seriously!) which was the first to claim evidence in support of "neutrino oscillations".

Theory had shown that a quantity R could be measured and that the (electron) neutrino was not stable (*i.e.* it oscillated) if $R \le 0.42$.



For this experiment it turned out that: $R = \dfrac{a}{\dfrac{d}{k^2 e}(b-c) - 2a\left(1 - \dfrac{kd}{e}\right)}$

The numerical values of the constants (and their uncertainties), as measured by this experiment were:

$$\begin{aligned} a &= 3.84 \pm 1.33 \\ b &= 74.0 \pm 4.0 \\ c &= 9.5 \pm 3.0 \\ d &= 0.112 \pm 0.009 \\ e &= 0.320 \pm 0.002 \\ k &= 0.89 \end{aligned}$$

If one ***assumes*** that the ***linear*** approximation in the Taylor series expansion for ***error propagation*** **is** valid, then one obtains: $R = 0.191 \pm 0.073 \; (\le 0.42)$, which is a $(0.420 - 0.191)/0.073 \simeq 3.1\sigma$ effect.

If we ***assume*** that ***all*** uncertainties ***are*** Gaussian/normally-distributed, then the probability that a measurement will be more than 3.1 standard deviations from its expectation value (*i.e.* a value of $R > 0.42$) is 0.001, or 0.1%. The authors thus claimed that their experimental result was strong evidence for neutrino oscillations…

However, note that the parameters a and c have very large uncertainties (on the order of 30%) and so *e.g.* $\sigma_a^2/a^2 \ll 1$ in the Taylor series expansion for ***error propagation*** is **not** satisfied. Recall from P598AEM Lect. Notes 5 that the "familiar" formula for the standard deviations of ***products*** xy and ***quotients*** x/y ***requires*** this to be so!

In such a case, a more ***correct*** procedure would be (assuming that the P.D.F.s of the measurements ***are*** indeed purely Gaussian) to *randomly* generate $a_i, b_i, c_i, d_i, e_i \quad (i = 1, N)$, each according to its own P.D.F. (using the central value and standard deviation measured in the experiment) and then calculate the corresponding value of R_i for each set of a_i, b_i, c_i, d_i, e_i.

If we actually count the # of entries beyond $R = 0.42$ in the above histogram, we discover that about 4% of the R values exceed 0.42. Thus, there is in fact a 4% (not 0.1%!!!) chance that a "stable neutrino" will yield the results of the experiment. This is why the first claim of evidence for neutrino oscillations obtained from measurements of endpoint of the tritium β-decay spectrum was not taken too seriously…

The use of MC **Propagation of Large Errors** techniques is potentially even more powerful than that described above. In the above example, suppose *e.g.* some (or even all) of the uncertainties on the *a-e* parameters in the *R*-expression are not exactly Gaussian/normally-distributed but in fact have "tails" on them.

By making a semi-log plot of each quantity *a-e*, there should be clear evidence for a Gaussian, parabola-shaped "core" to distribution for small σ-values, however for large σ-values, low-level "tails" of the distribution may arise *e.g.* from one or more unaccounted-for physics source(s), possibly having a Gaussian-type parabola-shape to it. In fact, a double-Gaussian LSQ fit of the uncertainty distribution for a common mean μ_x can easily be carried out of the form:

$$\alpha g_1\left(\mu_x, \sigma_{x_1}\right) + \beta g_1\left(\mu_x, \sigma_{x_2}\right)$$

where we expect $\sigma_{x_1} < \sigma_{x_2}$ and thus the normalization constants $\alpha > \beta$. If double-Gaussian LSQ fit results accurately described each of the above example's *a-e* variables, then the LSQ fit parameters could be used directly in an enhanced version of the above MC simulation of R to obtain an improved determination of the ***overall*** uncertainty on R – and more importantly, a P.D.F. for R from which an improved estimate of the probability for exceeding $R > 0.42$ could thus be obtained.

Furthermore, with the use of MC **Propagation of Large Errors** techniques, ***asymmetrical*** errors and/or ***non-Gaussian*** uncertainties associated with the *a-e* variables can also easily be incorporated in the MC simulation – even using *e.g.* actual ***histograms*** of the experimentally measured uncertainties on the *a-e* parameters as MC P.D.F.'s of the same.

Additionally, ***correlations*** between any/all of the *a-e* variables can also be accurately modeled via MC simulation as well.

Thus, it can be seen that the use of MC **Propagation of Large Errors** techniques in principle is accurate to ***all*** orders in the (infinite) Taylor series expansion that we use - only in the linear regime (*e.g.* $\sigma_a^2/a^2 \ll 1$) for ***error propagation***.

Potentially, MC **Propagation of Large Errors** techniques are ***very*** powerful, indeed!!!

Lots of useful MC techniques such as the ones discussed above ***and more*** (*e.g.* "tricks of the trade"!) are contained in "***A Third Monte Carlo Sampler***", LASL Informal Report LA-9721-MS, Los Alamos National Laboratory, by C.J. Everett and E.D. Cashwell (1983).

This document *e.g.* describes techniques for generating pairs or triplets, *etc*. of correlated Gaussians, and more!

See also *e.g.* the Monte Carlo Techniques section of the Review of Particle Physics, available on the web at the following URL: http://pdg.lbl.gov/

The following books also have much useful/helpful information on various MC techniques:

W. H. Press, *et al*., Numerical Recipes, 3rd Ed., Cambridge University Press, New York, 2007.

D.E. Knuth, The Art of Computer Programming, Vol. II, 3rd Ed., Addison-Wesley, Reading, MA 1998.