1. Monte Carlo integration

The most common use for Monte Carlo methods is the evaluation of integrals. This is also the basis of Monte Carlo simulations (which are actually integrations). The basic principles hold true in both cases.

1.1. Basics

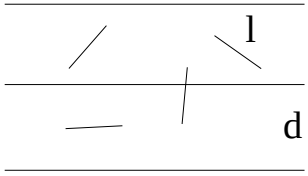
- Basic idea becomes clear from the "dartboard method" of integrating the area of an irregular domain:

  Choose points randomly within the rectangular box

$$\frac{A}{A_{\text{box}}} = p(\text{hit inside area}) \leftarrow \frac{\text{\# hits inside area}}{\text{\# hits inside box}}$$

  as the # hits $\rightarrow \infty$.

- Buffon's (1707–1788) needle experiment to determine π:
  - Throw needles, length l, on a grid of lines distance d apart.
  - Probability that a needle falls on a line:



$$P = \frac{2l}{\pi d}$$

  - Aside: Lazzarini 1901: Buffon's experiment with 34080 throws:

$$\pi \approx \frac{355}{113} = 3.14159292$$

    Way too good result!

    (See "π through the ages",

    http://www-groups.dcs.st-and.ac.uk/~history/HistTopics/Pi_through_the_ages.html)

1.2. Standard Monte Carlo integration

- Let us consider an integral in d dimensions:

$$I = \int_V d^d x f(x)$$

  Let V be a d-dim hypercube, with $0 \leq x_\mu \leq 1$ (for simplicity).

- Monte Carlo integration:

  - Generate N random vectors x_i from flat distribution ($0 \leq (x_i)_\mu \leq 1$).

  - As $N \to \infty$,

$$\frac{V}{N} \sum_{i=1}^{N} f(x_i) \to I \, .$$

  - Error: $\propto 1/\sqrt{N}$ independent of d! (Central limit)

- "Normal" numerical integration methods (Numerical Recipes):

  - Divide each axis in n evenly spaced intervals

- Total # of points $N \equiv n^d$
- Error:
  - $\propto 1/n$ (Midpoint rule)
  - $\propto 1/n^2$ (Trapezoidal rule)
  - $\propto 1/n^4$ (Simpson)

- If d is small, Monte Carlo integration has much larger errors than standard methods.
- When is MC as good as Simpson? Let's follow the error

$$1/n^4 = 1/N^{4/d} = 1/\sqrt{N} \qquad \rightarrow \qquad d = 8.$$

  In practice, MC integration becomes better when $d \gtrsim 6$–8.
- In lattice simulations $d = 10^6 - 10^8$!
- In practice, N becomes just too large very quickly for the standard methods: already at $d = 10$, if $n = 10$ (pretty small), $N = 10^{10}$. Too much!

1.3. Why error is $\propto 1/\sqrt{N}$?

- This is due to the *central limit theorem* (see, f.ex., L.E.Reichl, Statistical Physics).
- Let $y_i = Vf(x_i)$ (absorbing the volume in f) be the value of the integral using one random coordinate (vector) x_i. The result of the integral, after N samples, is

$$z_N = \frac{y_1 + y_2 + \ldots + y_N}{N} .$$

- Let $p(y_i)$ be the probability distribution of y_i, and $P_N(z_N)$ the distribution of z_N. Now

$$P_N(z_N) = \int dy_1 \ldots dy_N \; p(y_1) \ldots p(y_N) \delta(z_N - \sum_i y_i/N)$$

- Now

$$1 \;=\; \int dy \, p(y)$$

$$\begin{aligned} \langle y \rangle &= \int dy\, y\, p(y) \\ \langle y^2 \rangle &= \int dy\, y^2\, p(y) \\ \sigma &= \langle y^2 \rangle - \langle y \rangle^2 = \langle (y - \langle f \rangle)^2 \rangle \end{aligned}$$

Here, and in the following, $\langle \alpha \rangle$ means the expectation value of α (mean of the distribution $p_\alpha(\alpha)$).

- The error of the Monte Carlo integration (of length N) is *defined* as the width of the distribution $P_N(z_N)$, or more precisely,

$$\sigma_N^2 = \langle z_N^2 \rangle - \langle z_N \rangle^2 \,.$$

- Naturally $\langle z_N \rangle = \langle y \rangle$.
- Let us calculate σ_N:

- Let us define the Fourier transformation of $p(y)$:

$$\phi(k) = \int dy\; e^{ik(y - \langle y \rangle)}\, p(y)$$

Similarly, the Fourier transformation of $P_N(z)$ is

$$\begin{aligned}\Phi_N(k) &= \int dz_N\, e^{ik(z_N-\langle z_N\rangle)}\, P_N(z_N) \\ &= \int dy_1 \ldots dy_N\ \ e^{i(k/N)(y_1-\langle y\rangle+\ldots+y_N-\langle y\rangle)}\ \ p(y_1)\ldots p(y_N) \\ &= [\phi(k/N)]^N\end{aligned}$$

- Expanding $\phi(k/N)$ in powers of k/N (N large), we get

$$\phi(k/N) = \int dy\ e^{i(k/N)(y-\langle y\rangle)}\, p(y) = 1 - \frac{1}{2}\frac{k^2\sigma^2}{N^2} + \ldots$$

Because of the oscillating exponent, $\phi(k/N)$ will decrease as $|k|$ increases, and $\phi(k/N)^N$ will decrease even more quickly. Thus,

$$\Phi_N(k) = \left(1 - \frac{1}{2}\frac{k^2\sigma^2}{N^2} + O(\frac{k^3}{N^3})\right)^N \longrightarrow e^{-k^2\sigma^2/2N}$$

- Taking the inverse Fourier transform we obtain

$$\begin{aligned} P_N(z_N) &= \frac{1}{2\pi}\int dk\ e^{-ik(z_N-\langle y\rangle)}\Phi_N(k) \\ &= \frac{1}{2\pi}\int dk\ e^{-ik(z_N-\langle y\rangle)}e^{-k^2\sigma^2/2N} \\ &= \sqrt{\frac{N}{2\pi}}\frac{1}{\sigma}\exp\left[-\frac{N(z_N-\langle y\rangle)^2}{2\sigma^2}\right]. \end{aligned}$$

- Thus, the distribution $P_N(z_N)$ approaches Gaussian as $N \to \infty$, and

$$\sigma_N = \sigma/\sqrt{N}.$$

1.4. In summary: error in MC integration

If we integrate

$$I = \int_V dx\, f(x)$$

with Monte Carlo integration using N samples, the error is

$$\sigma_N = V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N}}\,.$$

In principle, the expectation values here are exact properties of the distribution of f, i.e.

$$\langle f\rangle = \frac{1}{V}\int dx f(x) = \int dy\, y\, p(y) \qquad \langle f^2\rangle = \frac{1}{V}\int dx f^2(x) = \int dy\, y^2 p(y).$$

Here $p(y)$ was the distribution of values $y = f(x)$ when the x-values are sampled with flat distribution. $p(y)$ can be obtained as follows: the probability of the x-coordinate is flat, i.e. $p_x(x) = C$ =const., and thus

$$p_x(x)dx = C\frac{dx}{dy}dy \equiv p(y)dy \qquad \Rightarrow$$

$$p(y) = C/(dy/dx) = C/f'(x(y))\,.$$

Of course, we don't know the expectation values beforehand! In practice, these are substituted by the corresponding Monte Carlo estimates:

$$\langle f \rangle \approx \frac{1}{N}\sum_i f(x_i) \qquad\qquad \langle f^2 \rangle \approx \frac{1}{N}\sum_i f^2(x_i)\,. \tag{1}$$

Actually, in this case we must also use

$$\sigma_N = V\sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N-1}}$$

because using (1) here would underestimate σ_N.

Often in the literature the real expectation values are denoted by $\langle\langle x\rangle\rangle$, as opposed to the Monte Carlo estimates $\langle x\rangle$.

The error obtained in this way is called the 1-σ error; i.e. the error is the width of the Gaussian distribution of

$$f_N = 1/N\sum_i f(x_i).$$

It means that the true answer is within $V\langle f\rangle \pm \sigma_N$ with $\sim 68\%$ probability. This is the most common error value cited in the literature.

This assumes that the distribution of f_N really is Gaussian. The central limit theorem guarantees that this is so, provided that N is large enough (except in some pathological cases).

However, in practice N is often *not* large enough for the Gaussianity to hold true! Then the distribution of f_N can be seriously off-Gaussian, usually skewed. More refined methods can be used in these cases (for example, modified jackknife or bootstrap error estimation). However, often this is ignored, because there just are not enough samples to deduce the true distribution (or people are lazy).

Example: convergence in MC integration

- Let us study the following integral:

$$I = \int_0^1 dx\, x^2 = 1/3\,.$$

- Denote $y = f(x) = x^2$ and $x = f^{-1}(y) = \sqrt{y}$.
- If we sample x with a flat distribution, $p_x(x) = 1$, $0 < x \leq 1$, the probability distribution $p_f(y)$ of $y = f(x)$ can be obtained from

$$p_x(x)dx = p_x(x)|\frac{dx}{dy}|dy \equiv p_f(y)dy \qquad \Rightarrow$$
$$p_f(y) = |\frac{dx}{dy}| = |1/f'(x)| = \frac{1}{2}x^{-1} = \frac{1}{2}y^{-1/2}$$

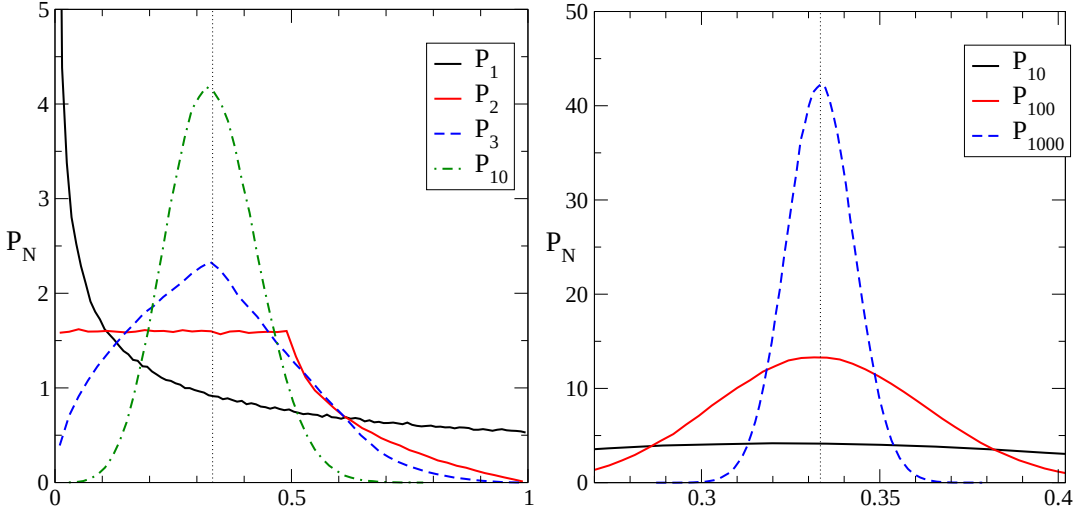
  where $0 < y \leq 1$.

- In more than 1 dim: $p_f(\vec{y}) = \left|\left|\frac{\partial x_i}{\partial y_j}\right|\right|$, the Jacobian determinant.

- The result of a Monte Carlo integration using N samples x_i, $y_i = f(x_i)$, is $I_N = (y_1 + y_2 + \ldots + y_N)/N$.
- What is the probability distribution P_N of I_N? This tells us how badly the results of the (fixed length) MC integration scatter.

$$
\begin{aligned}
P_1(z) &= p_f(z) = 1/\sqrt{z} \\
P_2(z) &= \int_0^1 dy_1 dy_2 \; p_f(y_1) p_f(y_2) \; \delta(z - \frac{y_1 + y_2}{2}) \\
&= \begin{cases} \pi/2 & 0 < y \leq 1/2 \\ \arcsin \frac{1-y}{y} & 1/2 < y \leq 1 \end{cases} \\
P_3(z) &= \int_0^1 dy_1 dy_2 dy_3 \; p_f(y_1) p_f(y_2) p_f(y_3) \, \delta(z - \frac{y_1 + y_2 + y_3}{3}) = \ldots \\
&\ldots
\end{aligned}
$$

P_N, measured from the results of 10^6 independent MC integrations for each N:



The expectation value = 1/3 for all P_N.

The width of the Gaussian is

$$\sigma_N = \sqrt{\frac{\langle\langle f^2\rangle\rangle - \langle\langle f\rangle\rangle^2}{N}} = \sqrt{\frac{4}{45N}}$$

where

$$\langle\langle f^2\rangle\rangle = \int_0^1 dx f^2(x) = \int_1^\infty dy y^2 p_f(y) = 1/5.$$

For example, using $N = 1000$, $\sigma_{1000} \approx 0.0094$. Plotting

$$C \exp\left[-\frac{(x-1/3)^2}{2\sigma_{1000}^2}\right]$$

in the figure above we obtain a curve which is almost undistinguishable from the measured P_{1000} (blue dashed) curve.

In practice, the expectation value and the error is of course measured from a single Monte Carlo integration. For example, using again $N = 1000$ and the Monte Carlo estimates

$$\langle f \rangle = \frac{1}{N} \sum_i f_i \,, \quad \langle f^2 \rangle = \frac{1}{N} \sum_i f_i^2 \,, \quad \sigma_N = \sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N-1}}$$

we obtain the following results (here 4 separate MC integrations):

1: 0.34300 ± 0.00950

2: 0.33308 ± 0.00927

3: 0.33355 ± 0.00931

4: 0.34085 ± 0.00933

Thus, both the average and the error estimate can be reliably calculated in a single MC integration.

Another example:

What about integral $I = \int_0^1 dx\, x^{-1/2} = 2$? Now $\langle\langle f^2\rangle\rangle = \int_0^1 dx\, x^{-1} = \infty$, diverges logarithmically. What happens to the MC error estimate?

The assumptions in the central limit theorem do not hold in this case. However, in practice the Monte Carlo integration works also now, and the distribution P_N approaches something like a Gaussian shape, and the characteristic width (and thus the error of the MC integration) behaves as $\propto 1/\sqrt{N}$. However, the standard formula with $\langle\langle f^2\rangle\rangle$ does not work.

Now

$$P_1(y) = p_f(y) = 1/f' = y^{-3},$$

where $1 \leq y < \infty$. Thus, $y^2 P_1(y)$ is not integrable, and since $P_2(y) \sim y^{-3}$ when $y \to \infty$, neither is $y^2 P_2(y)$. This remains true for any N(?). Thus, formally the error σ_N is infinite!

Nevertheless, we can perform standard MC integrations with MC estimates for $\langle f\rangle$, $\langle f^2\rangle$ and plug these into the formula for σ_N:

For example, some particular results

$N = 1000 : \quad 1.897 \pm 0.051$

$N = 10000 : \quad 1.972 \pm 0.025$

$N = 100000 : \quad 2.005 \pm 0.011$

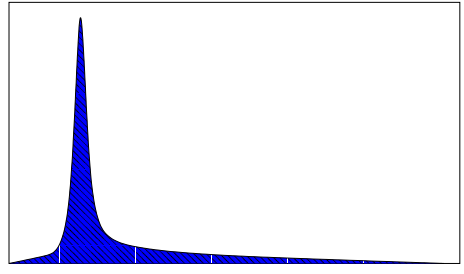
$N = 1000000 : \quad 1.998 \pm 0.003$

Results may fluctuate quite a bit, since occasionally there may occur some very large values of $f(x) = 1/\sqrt{x}$.

1.5. Importance sampling

Often the integrand has a very small value on a dominant fraction of the whole integration volume:

If the points are chosen evenly in the integration volume, the small minority of the points close to the “peak” give the dominant contribution to the integral.



Importance sampling: choose the random points so that more points are chosen around the peak, less where the integrand is small.

$\mapsto$ reduces the variance σ, and thus reduces the error.

Importance sampling is essential in Monte Carlo simulations! In that case we integrate $\int [d\phi] \exp[-S]$, which is exponentially strongly peaked function (peak near the corner where S is small).

- Let us integrate $I = \int_V dV\, f(x)$
- Choose a distribution $p(x)$, which is close to the function $f(x)$, but which is simple enough so that it is possible to generate random x-values from this distribution.
- Now

$$I = \int dV p(x) \frac{f(x)}{p(x)}\,.$$

- Thus, if we choose random numbers x_i from distribution $p(x)$, we obtain

$$I = \lim_{N\to\infty} \frac{1}{N} \sum_i \frac{f(x_i)}{p(x_i)}$$

- Since f/p is flatter than f, the variance of f/p will be smaller than the variance of $f \rightarrow$ error will be smaller (for a given N).

- Ideal choice: $p(x) \propto |f(x)|$. Now the variance vanishes! In practice not usually possible, since the proportionality constant is the unknown integral!
- However, exactly this choice is done in Monte Carlo simulations.

- In Monte Carlo simulations, we want to evaluate integrals of type

$$I = \int [d\phi]\, f(\phi)\, e^{-S(\phi)} / \int [d\phi]\, e^{-S(\phi)}$$

(using here notation $[d\phi] = \Pi_x\, d\phi$.)

- Importance sampling: we generate random $\phi(x)_i$ -configurations with probability

$$p(\phi) = \text{const.} \times e^{-S(\phi)}$$

(with unknown normalization constant).

- Applying the previous result:

$$I \approx \frac{1}{N} \sum_i f(\phi_i)$$

- Note that

$$Z = \int [d\phi]\, e^{-S(\phi)}$$

cannot be directly calculated with importance sampling, due to the unknown normalization constant.

- In a sense, this importance sampling turns the plain Monte Carlo integration into simulation: the configurations $\phi(x)$ are sampled with the Bolzmann probability $\propto e^{-S}$, which is the physical probability of a configuration. Thus, the integration process itself mimics nature!
- How to generate ϕ's with probability $\propto e^{-S(\phi)}$? That is the job of the Monte Carlo update algorithms.

Example: importance sampling

Integrate

$$I = \int_0^1 dx\,(x^{-1/3} + x/10) = 31/20 = 1.55$$

Standard MC gives error

$$\sigma_N = \sqrt{\frac{\langle f^2\rangle - \langle f\rangle^2}{N-1}} \approx \frac{0.85}{\sqrt{N-1}}$$

Let us now assume that we don't know the function but we can evaluate it. We recognize that it diverges pretty much like $x^{-1/3}$ as x is small. Let us therefore propose to do the integral with importance sampling, using sampling probability

$$p(x) = 2/3x^{-1/3}, \qquad 0 < x \leq 1.$$

How to obtain random numbers with distribution $p(x)$? (This will be discussed in detail in the next section.) Let us make a change of variable

so that the distribution in the new variable is flat:

$$p(x)dx = p(x)\frac{dx}{dy}dy = dy \qquad \Rightarrow \qquad y = \int_0^x dx'\, p(x') = x^{2/3}$$

Thus, if y is from a flat distribution $y \in (0, 1]$, $x = y^{3/2}$ is from distribution $p(x) = 2/3x^{-1/3}$.
In the new variable y the integral becomes

$$I = \int_0^1 dx p(x)\frac{f(x)}{p(x)} = \int_0^1 dy \frac{f(x(y))}{p(x(y))}$$

Thus, if we denote $g = f/p$, we have $\langle g \rangle = 31/20$ and $\langle g^2 \rangle = 2.4045$. Thus, the width of the result in MC integration of g is

$$\sigma_N = \sqrt{\frac{\langle g^2 \rangle - \langle g \rangle^2}{N-1}} \approx= \frac{0.045}{\sqrt{N-1}}$$

This is $\sim$ 20 times narrower than with the naive method!

The recipe for the MC integration with importance sampling is thus:

- generate N random numbers y_i from flat distribution
- $x_i = y_i^{3/2}$ are from distribution $p(x)$
- calculate the average of $g_i = f(x_i)/p(x_i)$

Indeed:

N	naive	importance
100	1.4878 ± 0.0751	1.5492 ± 0.0043
10000	1.5484 ± 0.0080	1.5503 ± 0.0004

1.6. Note: some standard MC integration routines

Powerful Monte Carlo integration routines are included in several numerical packages. However, typically these tend to fail when number of dimensions is larger than ~ 15. Thus, these are useless for Monte Carlo simulations.
The following routines are included in the GSL (GNU scientific library, available for free). See also Numerical Recipes 7.8 for further details.

- VEGAS: uses approximate importance sampling (together with several other tricks). It does several passes over the integral, and uses the results of the previous passes to improve its estimate of the importance sampling function.

  Described in:

  G.P. Lepage, 'A New Algorithm for Adaptive Multidimensional Integration', Journal of Computational Physics 27, 192-203, (1978)

- MISER: uses *stratified sampling*

1.7. Note: quasirandom sequences

- Quasirandom sequences are “random” number sequences which are not random at all, but are (for our purposes) specially constructed to cover the large-dimensional volume (hypercube) as evenly as possible.
- Error in quasirandom integration usually $\propto 1/N$ or even faster, where N is the number of points in the sequence (cf. $1/\sqrt{N}$ in random Monte Carlo).
- Sequences depend on d and N.
- *Does not work with importance sampling $\rightarrow$ not useful for Monte Carlo simulation.*
- Numerical Recipes sec. 7.7; implementations in GSL (GNU scientific library, available for linux and other systems).

2. Random numbers

Good random numbers play a central part in Monte Carlo simulations. Usually these are generated using a deterministic algorithm; the numbers generated this way are called *pseudorandom numbers*.

There are several methods for obtaining random numbers for Monte Carlo simulations, however, there have been occasions where the random numbers generated with a trusted old workhorse algorithm have failed (i.e. the simulation has produced incorrect results). What is understood to be a “good” random number generator varies with time!

2.1. Physical random numbers

Physical random numbers are generated from some truly random physical process (radioactive decay, thermal noise, roulette wheel . . .). Before the computer age, special machines were built to produce random numbers which were often published in books. For example, 1955 the RAND

corporation published a book with a million random numbers, obtained using an electric “roulette wheel”. This classic book is now available on the net, at

`http://www.rand.org/publications/classics/randomdigits/`

Recently, the need for really random (non-algorithmic) numbers in computer technology has increased dramatically. This is due to the wide use of *cryptography* (ssh, ipv6, encrypted files/disks, . . .).

For example, in Linux the special character files `/dev/random` and `/dev/urandom` return random bytes (characters) when read from a program. For example, you can try

`more /dev/random`

which will print gibberish on the screen (may lock the terminal, since some characters are not printable!). This interface generates randomness, “entropy pool”, by snooping various physical timings etc. from device drivers (key press intervals, mouse pointer movements, disk access times, internet packet delays etc.).

More info with command `man 4 random`.

Due to the increase in cryptography, some modern processors also introduce a hardware random number generator (Intel P4). Some very old computers had hardware random numbers too, but those were meant mostly for Monte Carlo use (with questionable success).

Physical random numbers are not very useful for Monte Carlo, because:

- The sequence is not repeatable.
- The generators are often slow.
- The quality of the distribution is often not perfect. For example, a sequence of random bits might have slightly more 0's than 1's. This is not so crucial for cryptography (as long as the numbers are really random), but is absolute no-no for Monte Carlo.

Silicon Graphics' *Lavarand*: generate random numbers from the digital tv-camera images of a lava lamp.

2.2. Pseudorandom numbers

Almost all of the Monte Carlo calculations utilize *pseudorandom numbers*, which are generated using deterministic algorithms. Typically the generators produce a random integer (with a definite number of bits), which is converted to a floating point number number $X \in [0, 1)$ or $[0, 1]$ by multiplying with a suitable constant.

The generators are initialized once before use with a *seed number*, typically an integer value or values. This sets the initial state of the generator.

The essential properties of a good random number generator are:

Repeatability – the same initial values (seeds) produces the same random number sequence. This can be important for debugging.

Randomness – random numbers should be
a) from *uniform* distribution – for example, really homogeneously distributed between [0,1)

b) *non-correlated*, i.e. independent of each other. This is tough! No pseudorandom sequence is truly independent.

Long period – the generators have a finite amount of internal state information, so the sequences must repeat itself after finite period. The period should be much longer than the amount of numbers needed for the calculation (preferably).

Insensitive to seeds – the period and randomness properties should not depend on the initial seed.

Fast

Portability – same results on different computers.

One form of pseudorandom numbers are sequences extracted from the numerical representations of π or other transcendental numbers (studied by J. Von Neumann and N. Metropolis 1950's). These are not very practical, however.

Let us now look at some main types of pseudorandom number generators.

2.2.1. Midsquare method

This is only of historical note, it is the first pseudorandom generator (N. Metropolis). Don't use it for any serious purpose. This works as follows: take a n-digit integer; square it, giving a $2n$-digit integer. Take the middle n digits for the new integer value.

2.2.2. Linear congruential generator

One of the simplest, widely used and oldest (Lehmer 1948) generators is the linear congruential generator (LCG). Usually the language or library "standard" generators are of this type.
The generator is defined by integer constants a, c and m, and produces

a sequence of random integers X_i via

$$X_{i+1} = (aX_i + c) \bmod m$$

This generates integers from 0 to $(m-1)$ (or from 1 to $(m-1)$, if $c = 0$). Real numbers in $[0, 1)$ are obtained by division $f_i = X_i/m$.

Since the state of the generator is specified by the integer X_i, which is smaller than m, the period of the generator is *at most m*. The constants a, c and m must be carefully chosen to ensure this. Arbitrary parameters are sure to fail!

These generators have by now well-known weaknesses. Especially, if we construct d-dimensional vectors ("d-tuples") from consecutive random numbers $(f_i, f_{i+1}, \ldots, f_{i+d})$, the points will lie on a relatively small number of hyperplanes (at most $m^{1/d}$, but can be much less; see Numerical Recipes).

In many generators in use $m = 2^n$. In this case, it is easy to see that the low-order bits have very short periods. This is due to the fact that the information in X is only moved "up", towards more significant bits, never

down. Thus, for k least significant bits there are only 2^k different states, which is then the cycle time of these bits. The lowest order bit has a period of 2, i.e. it flips in sequence 101010.... Some amount of cycling occurs in fact always when m is not a prime.

Thus, if you need random integers or random bits, *never use the low-order bits from LGC's*!

- The ANSI C standard[1] random number routine `rand()` has parameter values

  $$a = 1103515245,\ c = 12345,\ m = 2^{31}$$

  This is essentially a 32-bit algorithm. The cycle time of this generator is only $m = 2^{31} \approx 2 \times 10^9$, which is exhausted very quickly in a modern computer. Moreover, m is a power of 2, so that the

[1] Sorry: not standard, but *mentioned* in the standard. The standard does not specify a specific generator.

low-order bits are periodic. In Linux, function `rand()` has been substituted by a more powerful function.

This routine exists in many, maybe most, system libraries (and may be the random number in Fortran implementations). Nevertheless this generator is not good enough for serious computations.

- GGL, IBM system generator (Park and Miller "minimal standard")

  $$a = 16807,\ c = 0,\ m = 2^{31} - 1$$

  As before, short cycle time. Better generator than the first one, but I would not recommend this for Monte Carlo simulations. This generator is the RAND generator in MATLAB.

- UNIX `drand48()`:

  $$a = \mathrm{5DEECE66D}_{16},\ c = \mathrm{B}_{16},\ m = 2^{48}$$

  This uses effectively 64 bits in arithmetics, and the internal state is modded to a 48-bit integer. The cycle time is thus $2^{48} \approx 2.8 \times 10^{14}$.

The cycle time is sufficient for most purposes, and this generator is much used. However, it has the common low-order cycling problem and must be considered pretty much obsolete.

- NAG (Numerical Algorithms Group):

$$a = 13^{13},\ c = 0,\ m = 2^{59}$$

Very long cycle time, with low-order cycling.

2.2.3. Lagged Fibonacci & friends

Lagged Fibonacci and related generators improve the properties of the random numbers by using much more than one integer as the internal state. Generally, lagged Fibonacci generators form a new integer X_i using

$$X_i = (X_{i-p} \odot X_{i-q}) \bmod m$$

where p and q are *lags*, integer constants, $p < q$, and $\odot$ is some arithmetic operation, such as +, -, * or $\oplus$, where the last is XOR, exclusive bitwise or.

The generator must keep at least q previous X_i's in memory. The quality of these generators is not very good with small lags, but can become excellent with large lags. If the operator $\odot$ is addition or subtraction, the maximal period is $\sim 2^{p+q-1}$.

If the operator $\odot$ is XOR, the generator is often called (generalized) feedback shift register (GFSR) generator. Standard UNIX `random()` gener-

ator is of this type, using up to 256 bytes of internal storage. I believe this is the `rand()` in Linux distributions. I *don't* recommend either for Monte Carlo simulations.
Another fairly common generator of this type is R250:

$$X_i = X_{103} \oplus X_{250}$$

This requires 250 integer words of storage (For more information, see Vattulainen's web-page). However, GFSR generators are known to have rather strong 3-point correlations in tuples (X_i, X_{i-q}, X_{i-p}) (no surprise there). It has been observed to fail spectacularly in Ising model simulations using a Wolff cluster algorithm (Ferrenberg et al., Phys. Rev. Lett. 69 (1992)). One should probably not use these kind of generators in serious Monte Carlo.

That said, these generators typically do not distinguish between low- and high-order bits; thus, the low-order bits tend to be as good as the high order ones.

Because of the large internal memory, seeding of these generators is somewhat cumbersome. Usually one uses some simpler generator, e.g. some LGC, to seed the initial state variables.

2.2.4. Mersenne twister

Mersenne twister or MT19937 is modern very powerful generator (Matsumoto & Nishimura 1997). It is a modification of a genralized feedback shift register, and it uses 624 32-bit integers as the internal storage. It uses special bit shuffling and merging in addition to the XOR operation of the standard shift register generator.

The period is huge, exactly $2^{19937} - 1$, and the spectral properties are also proven to be very good up to 623 dimensions. The generator is also fairly fast (especially if one inlines some parts of it).

Home page: `http://www.math.keio.ac.jp/~matumoto/emt.html`

This generator has been incorporated into many packages/languages. It should be a good generator for Monte Carlo simulations.

2.2.5. Combined generators

Many of the bad properties of single random number generators can be avoided by combining two (or more) generators. For example,

$$\begin{aligned} x_i &= (40014 x_{i-1}) \bmod 2147483563 \\ y_i &= (40692 y_{i-1}) \bmod 2147483399 \\ X_i &= (x_i + y_i) \bmod 2147483563 \end{aligned}$$

forms the core of the combined generator by l'Ecuyer and Bays-Durham, presented in Numerical Recipes as `ran2`; the generator also implements some modifications to the above simple procedure; shuffling of the registers etc. The period of this is the product of the mod-factors, $\sim 10^{18}$.

RANMAR, by Marsaglia, Zaman and Tsang, is a famous combined generator of a lagged Fibonacci and a LGC generator with a prime modulus $m = 2^{24} - 3$. Period is good, 2^{144}, but it uses only 24 bits as the working precision (single precision reals). This generator has passed all the tests

thrown at it.

RANLUX, by Lüscher, is a lagged Fibonacci generator with adjustable skipping, i.e. it rejects a variable number of random number candidates as it goes. It produces “luxurious” random numbers with mathematically proven properties. In higher luxury levels it becomes somewhat slow, however, even at the smallest luxury level the period is about 10^{171}. Luxury level 3 is the default, luxury level 4 makes all bits fully chaotic. The code is pretty long, though. (Computer physics communications 79 (1994) 100).

CMRG, combined multiple recursive generator by l’Ecuyer (Operations Research 44,5 (1996)) (one of the several generators by L’Ecuyer). It uses 6 words of storage for the internal state, and has a period of $\approx 2^{205}$.

2.2.6. Which generator to use?

The generators must be tested in order to separate the wheat from the chaff. A well-known test suite is the DIEHARD test by Marsaglia, available from the net (but it contains bugs, apparently...). However, the generators that bombed in the 90's had passed (almost?) all of the "synthetic" tests, but were found lacking in real Monte Carlo simulations. For further info, see, for example, Vattulainen's web page.

- Mersenne twister: probably good overall, fast. My current favourite. Code available.
- RANLUX: very good, somewhat slower at high (= good enough) luxury levels.
- RANMAR: already very well established, and no problems so far.
- `drand48`: part of the standard UNIX library, thus immediately available if you don't have anything else. Good enough for most uses, but don't generate random bit patterns with it!

Practical aspects

- Never use a black box
- Never try to “improve” any generator by fiddling the parameters
- Take care with initialization (seeding)
- Use a well-known and tested generator
- Test your results with 2 different generators!

Information about random number generators:

– Numerical Recipes

– Ilpo Vattulainen’s random number tests, see `http://www.physics.helsinki.fi/~vattulai/rngs.html`,

– pLab: `http://random.mat.sbg.ac.at/`

– D. Knuth: *The Art of Computer Programming, vol 2: Seminumerical Methods*

– P. L’Ecuyer, *Random numbers for simulation*, Comm. ACM 33:10, 85 (1990)

– L’Ecuyer’s home page: `http://www.iro.umontreal.ca/~lecuyer`

2.2.7. Using rng's

Example: drand48 generator in UNIX, from program written in C:

```
#include <stdio.h>       /* contains standard I/O definitions */
#include <stdlib.h>      /* std library, including drand */
#include <math.h>        /* contains normal functions */

int main()
{
   long seed;
   int i;
   double d;

   printf("Give seed: ");
   scanf("%ld",&seed);
   srand48( seed );    /* seed the generator */

   for (i=0; i<5; i++) {
      d = drand48();       /* get the random number */
      printf("%d  %g  %.10g\n", i, d, exp(d) );
   }
   return(1);
}
```

To compile: `cc -O3 -o prog prog.c -lm`

And the output is:

```
gluon(~/tmp)% ./prog
Give seed: 21313
0  0.580433  1.786811284
1  0.686466  1.986682984
2  0.586646  1.797948229
3  0.515342  1.674211124
4  0.783321  2.188729827
```

Using (my inline version of) the Mersenne twister: you need the files `mersenne_inline.c` and `mersenne.h`, given in course www-page. "The official" Mersenne twister code, also available in fortran, is available at the twister home page.

```
 ...
#include "mersenne.h"
 ...
int main()
{
   long seed;
   ...
   seed_mersenne( seed );
   d = mersenne();
   ...
```

Compile: `cc -O3 -o prog prog.c mersenne_inline.c -lm`

The inline version actually substitutes the function call `mersenne()` with a small piece of code. No function call → faster execution.

2.2.8. Note about implementing LGC's

Let us consider linear congruential generator

$$X_{i+1} = (aX_i) \bmod m$$

On a typical generator, m is of order 2^{31} or larger; thus, X_i can also be of the same magnitude. This means that the multiplication aX_I will overflow typical 32-bit integer (for example, `int` and `long` on standard intel PC's).

If m is a power of 2, this is not a problem (at least in C): if a and X are of type `unsigned int` or `unsigned long`, the multiplication gives the low-order bits correctly (and just drops the high-order bits). Modding this with a power of 2 gives then a correct answer.

However, if m is not a power of 2, this does not work.

• The easiest solution is to use 64-bit double precision floating points for the numbers. The mantissa on double (IEEE) has ~ 52 bits, so integers smaller than 2^{51} can be represented exactly.

• Or, sticking with ints, one can use the Shrage's algorithm (Numerical Recipes): we can always write m as $m = aq + p$, where $q = [m/a]$, the integer part of m/a. Now it is easy to show that

$$(aX) \bmod m = a(X \bmod q) - [X/q] \ \ \{+m\}$$

where $\{+m\}$ is added, if needed, to make the result positive.

2.3. Random numbers from non-uniform distributions

The pseudorandom generators in the previous section all return a random number from uniform distribution $[0, 1)$ (or $(0, 1)$ or some other combination of the limits). However, usually we need random numbers from non-uniform distribution. We shall now discuss how the raw material from the generators can be transformed into desired distributions.

2.3.1. Exact inversion

In general, probability distributions are functions which satisfy

$$\int dx p(x) = 1, \qquad p(x) \geq 0 \quad \text{for all } x.$$

Here the integral goes over the whole domain where $p(x)$ is defi ned.

The fundamental transformation law of probabilities is as follows: if we have a random variable x from a (known) distribution $p_1(x)$, and a function $y = y(x)$, the probability distribution of y, $p_2(y)$, is determined through

$$p_1(x)|dx| = p_2(y)|dy| \qquad \Rightarrow \qquad p_2(y) = p_1(x) \left| \frac{dx}{dy} \right|$$

In more than 1 dimensions: $|\frac{dx}{dy}| \to ||\frac{dx_i}{dy_j}||$, the Jacobian determinant of the transformation.

Now we know the distribution $p_1(x)$ and we also know the desired distribution $p_2(y)$, but we don't know the transformation law $y = y(x)$! It can be solved by integrating the above differential equation:[2]

$$\int_{a_1}^{x} dx' p_1(x') = \int_{a_2}^{y} dy' p_2(y') \quad \Leftrightarrow \quad P_1(x) = P_2(y) \quad \Leftrightarrow \quad y = P_2^{-1}[P_1(x)]\,,$$

where $P_1(x)$ is the *cumulant* of the distribution $p_1(x)$. a_1 and a_2 are the smallest values where $p_1(x)$ and $p_2(y)$ are defi ned (often $-\infty$).

Now $p_1(x) = 1$ and $x \in [0, 1]$. Thus, $P_1(x) = x$, and y is to be "inverted" from the equation

$$x = \int_{a_2}^{y} dy' p(y').$$

This is the fundamental equation for transforming random numbers from the uniform distribution to a new distribution $p(y)$. (Now I drop the subscript 2 as unnecessary.) Unfortunately, often the integral above is not very feasible to calculate, not to say anything about the fi nal inversion (analytically or numerically).

[2]Dropping the absolute values here means that $y(x)$ will be monotonously increasing function. We get monotonously decreasing $y(x)$ by using $\int_y^{b_2} dy'$ on the RHS.

Exponential distribution

Normalized exponential distribution for $y \in [0, \infty)$ is $p(y) = exp(-y)$. Thus, now the transformation is

$$x = \int_0^y dy' e^{-y'} = 1 - e^{-y} \qquad \Rightarrow \qquad y = -\ln(1-x) = -\ln x$$

We can use x or $1 - x$ above because of the uniform distribution; one should choose the one which does not ever try to evaluate $\ln 0$.

However, already the log-distribution $p(y) = -\ln y$, $0 < y \leq 1$ is not invertible analytically:

$$x = -\int_0^y dy' \ln y' = y(1 - \ln y)$$

This actually can be very effi ciently inverted to machine precision using *Newton's method* (Numerical Recipes, for example).

Gaussian distribution

One of the most common distributions in physics is the Gaussian distribution

$$p(y) = \frac{1}{\sqrt{2\pi}} e^{-y^2/2}.$$

The integral of this gives the error function. While `erf()` exists in many C libraries, it is not part of a standard. Using `erf()` and the Newton's method the transformation can be easily inverted. However, this is not very effi cient, and it is customary to use the following method.

The **Box-Muller method** generates Gaussian random numbers using 2-dimensional Gaussian distributions:

$$p(x,y) = \frac{1}{2\pi} e^{-(x^2+y^2)}.$$

This is, of course, only a product of 2 1-dimensional distributions. We can transform to polar coordinates $(x,y) \to (r,\theta)$

$$dxdy\, p(x,y) = dxdy\, \frac{1}{2\pi}\, e^{-(x^2+y^2)/2} = drd\theta\, r\, \frac{1}{2\pi}\, e^{-r^2/2} = drd\theta\, p(r,\theta).$$

Here r at the second stage is just the Jacobian of the transformation, as in the transformation law of probabilities. Since $p(r,\theta)$ factorizes to $p(r)p(\theta)$ (trivially, since it does not depend on θ), it is easy to transform two uniform random numbers (X_1, X_2) to (r,θ):

$$X_1 = \frac{1}{2\pi}\int_0^\theta d\theta' = \theta/2\pi, \qquad\qquad X_2 = \int_0^r dr' r e^{-r^2/2} = 1 - e^{-r^2/2}.$$

Inverting this, we get

$$\theta = 2\pi X_1, \qquad\qquad r = \sqrt{-2\ln X_2}.$$

These are finally converted to (x, y)-coordinates

$$x = r\cos\theta, \qquad y = r\sin\theta.$$

Both x and y are good Gaussian random numbers, so we can use both of them, one after another: on the first call to generator we generate both x and y and return x, and on the second call just return y.

This process implements two changes of random variables: $(X_1, X_2) \to (r, \theta) \to (x, y)$. On the second stage we did not have to do the integral inversion, because we knew the transformation from (x, y) to (r, θ).

It is customary to accelerate the algorithm above by eliminating the trigonometric functions. Let us first observe that we can interpret the pair $(\sqrt{X_2}, \theta)$ as the polar coordinates of a random point from a *uniform distribution inside a unit circle*. Why $\sqrt{X}$? This is because of the Jacobian; the uniform differential probability inside a circle is $\propto d\theta dr r$, which, when plugged in the conversion formula and integrated wrt. r yields $X_2 = r^2$.

Thus, instead of polar coordinates, we can directly generate cartesian coordinates from a uniform distribution inside a circle using the *rejection method*:

1. generate 2 uniform random numbers $v_i \in (-1, 1)$
2. accept if $R^2 = v_1^2 + v_2^2 < 1$, otherwise back to 1.

Now R^2 corresponds to X_2 above, and, what is the whole point of this transformation, $v_1/R \leftrightarrow \cos\theta$ and $v_2/R \leftrightarrow \sin\theta$. We don't have to evaluate the trigonometric functions.

```
/*****************  gaussian_ran.c ***************
 *  double gaussian_ran()
 *  Gaussian distributed random number
 *  Probability distribution exp( -x*x/2 ), so < x^2 > = 1
 *  Uses mersenne random number generator
 */
#include <stdlib.h>
#include <math.h>
#include "mersenne.h"

double gaussian_ran()
{
  static int iset=0;
  static double gset;
  register double fac,r,v1,v2;

  if  (iset) {
    iset = 0;
    return(gset);
  }

  do {
    v1 = 2.0*mersenne() - 1.0;
    v2 = 2.0*mersenne() - 1.0;
    r  = v1*v1 + v2*v2;
  } while (r >= 1.0 || r == 0.0);
  fac  = sqrt( -2.0*log(r)/r );
  gset = v1*fac;
  iset = 1;
  return(v2*fac);
}
```

2.3.2. Rejection method

The inversion method above is often very tricky. More often than not the function is not analytically integrable, and doing the integration + inversion numerically is expensive.

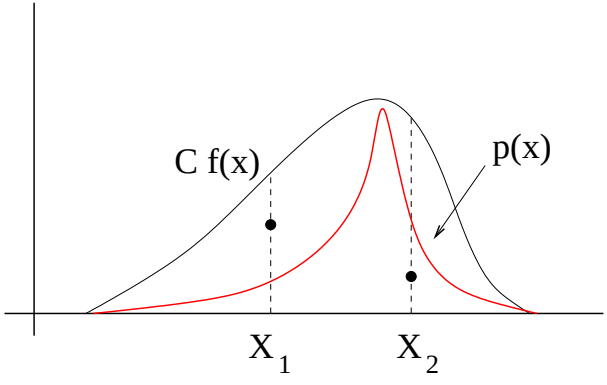
The *rejection method* is very simple and powerful method for generating numbers from any distribution. Let now $p(x)$ be the desired distribution, and let $f(x)$ be a distribution according to which we *know* how to generate random numbers, with the following property:

$$p(x) \leq C\, f(x)$$

with some known constant C. It is now essential to note that if we have a uniform density of points on a (x, y)-plane, the number of points in the interval $0 < y < C\, f(x)$ is proportional to $f(x)$. Same is true with $p(y)$. Now the method works as follows:

1. **Generate X from distribution $f(x)$.**
2. **Generate Y from uniform distribution $0 < Y < C\, f(X)$.** Now the point (X, Y) will be from an uniform distribution in the area below curve $C\, f(x)$.
3. **If $Y \leq p(x)$ return X.** This is because now the point (X, Y) is also a point in the uniform distribution below the curve $p(x)$, and we can interpret point X as being from distribution $p(x)$.
4. **If $Y > p(x)$ reject and return to 1.**

In the fi gure (X_1, Y_1) is rejected, but (X_2, Y_2) accepted. X_2 is thus a good random number from distribution $p(x)$.



- The rejection rate = (area under $p(x)$) / (area under $C\,f(x)$). Thus, $C\,f(x)$ should be as close to $p(x)$ as possible to keep the rejection rate small. However, it should also be easy to generate random variables with distribution $f(x)$.
- Often it is feasible to use $f(x) =$ const. (fast), but beware that the rejection rate stays tolerable.
- Works in any dimension.
- There's actually no need to normalize $p(x)$.

Example:

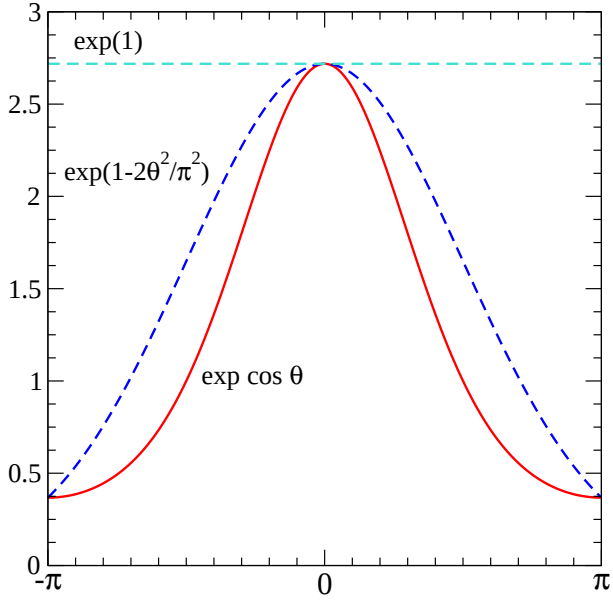
Consider (unnormalized) distribution

$$p(\theta) = \exp\cos\theta, \qquad -\pi < \theta < \pi\,.$$

This is not easily integrable, so let's use rejection method.

A) Generate random numbers from uniform distribution
$f(\theta) = e^1 \geq p(\theta)$.
Acceptance rate ≈ 0.46.

B) Generate random numbers from Gaussian distribution
$f(\theta) = \exp[1 - 2\theta^2/\pi^2] \geq p(\theta)$.
Acceptance rate ≈ 0.78



Thus, using B) we generate only $0.46/0.78 \approx 60\%$ of the random numbers used in A). Which one is faster depends on the speed of the random number generators.

2.3.3. Random numbers on and in a sphere

The trick used in the Box-Muller method to generate random numbers inside unit circle was a version of the rejection method. It can actually be used to generate random numbers inside or on the surface of a sphere in arbitrary dimensions:

1. Generate d uniform random numbers $X_i \in (-1, 1)$.
2. If $R^2 = \sum_i X_i^2 > 1$ the point is outside the sphere and go back to 1.
3. Otherwise, we now have a point $\vec{X}$ from an uniform distribution inside a d-dimensional spherical volume.
4. If we want to have a point on the *surface* of the sphere, just rescale $X_i \leftarrow X_i/R$. Now $\vec{X}$ will be evenly distributed on the surface.

The rejection rate is fairly small, unless the dimensionality is really large (homework). This method is almost always faster *and easier* than trying to use the polar coordinates (careful with the Jacobian!).

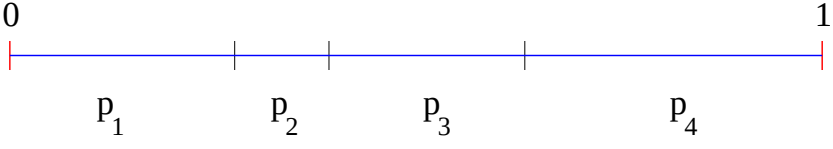
2.3.4. Discrete distributions

Very often in Monte Carlo simulations we need random numbers from discrete distributions (Ising model, Potts model, any other discrete variable model). These distributions are easy to generate:

Let i be our discrete random variable with N states (N can be ∞), and let p_i be the desired probability distribution. These are normalized

$$\sum_i p_i = 1 \, .$$

Imagine now a line which has been split into segments of length p_i:



Now, if we generate a uniform random number X, $0 < X < 1$, we get the discrete random number i by checking on which segment X lands. If N is small, this is easy to do by just summing $p_1 + p_2 \ldots + p_i$ until the sum is larger than X. For large N more advanced search strategies should be used (binary search, for example).

2.3.5. Redux: inversion when $p(x)$ is integrable

Sometimes the probability distribution $p(x)$ we want is relatively easily integrable, but not so easily invertible. Previously, we had the example $p(x) = -\ln x$, where $0 < x \leq 1$. The inversion formula is

$$X = -\int_0^x dy p(y) = P(x) = x(1 - \ln x)$$

where $X \in [0, 1]$ is a uniform random number.

Note that the cumulant $P(x)$ is always a monotonously increasing function and $0 \leq P(x) \leq 1$, so the equation *always* has an unique solution, no matter what $p(x)$.

A fast way to solve for x is the **Newton's method:**

1. Take initial guess for x, say $x_0 = X$.
2. Expand: $X \approx P(x_0) + (x - x_0)P'(x_0)$ and solve x:

$$x = x_0 + \frac{X - P(x_0)}{P'(x_0)} = x_0 + \frac{X - x_0(1 - \ln x_0)}{-\ln x_0}$$

3. If $|x - x_0| > \epsilon$ and/or $|X - P(x)| > \epsilon'$, where ϵ and ϵ' are required accuracy, set $x_0 = x$ and go back to 2.

4. Otherwise, accept x.

Convergence is usually very fast. However, beware the edges! x can easily wind up outside the allowed interval ($(0, 1)$ in this case).

In this example $y \equiv X$, the number from 0 to 1, and epsilon is the desired accuracy:

```
x = y;                                  /* initial value */
df = y - x*(1.0 - log(x));              /* y - P(x) */
while ( fabs(df) > epsilon ) {
  x1 = x - df/log(x);                   /* x - (y-P(x))/P'(x) */

  /* Check that x1 is within the allowed region! */
  if (x1 <= 0) x1 = 0.5*x;
  else if (x1 > 1) x1 = 0.5*(1.0 + x1);

  x  = x1;
  df = y - x*(1.0 - log(x));
}
```

Another option is the **binary search**:

1. Set x_1 and x_2 to minimum and maximum allowed x (0,1)
2. Set $x = (x_1 + x_2)/2$
3. If $P(x) < X$, set $x_1 = x$, otherwise $x_2 = x$
4. If $x_2 - x_1 > \epsilon$ go to 2.
5. Otherwise, accept x

This converges exponentially, factor of 2 for each iteration. The condition in 3. works because $P(x)$ increases monotonically.