The Monte Carlo method

A.Pompili (UniBA)

Statistical Data Analysis course

Bibliography:

- L.Lista, Statistical Methods for Data Analysis in Particle Physics, Lectures Notes in Physics 941, 2017, Springer Int. Pub.
- G.Cowan, Statistical Data Analysis, 1998, Oxford Science Publications.

Pseudorandom numbers

Many computer applications, ranging from simulations to video games and 3D- graphics, take advantage of computer-generated numeric sequences that have properties very similar to truly random variables.
Sequences generated by computer algorithms through mathematical operations are not really random, having no intrinsic unpredictability, and are necessarily *deterministic* and *reproducible*.
Indeed, the possibility to reproduce exactly the same sequence of computer-generated numbers with a computer algorithm can be a useful feature for many application.
Good algorithms that generate 'random' numbers or, more precisely, ***pseudorandom* numbers**, given their reproducibility, must obey, in the limit of large numbers, to the desired statistical properties of real random variables, with the limitation that pseudorandom sequences can be large, but not infinite.

Pseudorandom numbers

Many computer applications, ranging from simulations to video games and 3D- graphics, take advantage of computer-generated numeric sequences that have properties very similar to truly random variables.
Sequences generated by computer algorithms through mathematical operations are not really random, having no intrinsic unpredictability, and are necessarily *deterministic* and *reproducible*.
Indeed, the possibility to reproduce exactly the same sequence of computer-generated numbers with a computer algorithm can be a useful feature for many application.
Good algorithms that generate 'random' numbers or, more precisely, ***pseudorandom* numbers**, given their reproducibility, must obey, in the limit of large numbers, to the desired statistical properties of real random variables, with the limitation that pseudorandom sequences can be large, but not infinite.

Considering that computers have finite machine precision, pseudorandom numbers, in practice, have discrete possible values, depending on the number of bits used to store floating point variables (namely variables that can hold a real number [*]).

[*] https://www.freecodecamp.org/news/floating-point-definition/

Numerical methods involving the repeated use of computer-generated pseudo-random numbers are also known as…
… ***Monte Carlo* methods**, from the name of the city hosting the famous casino, where the properties of (truly) random numbers resulting from roulette and other games are exploited in order to generate profit.

Note: in the following we will refer to *pseudorandom* numbers simply as random numbers.

Uniform random number generators

The Monte Carlo method is a numerical technique for …
… calculating probabilities and related quantities by using sequences of random numbers.

The most widely used computer-based random number generators are conveniently written …
.. in order to produce sequences of uniformly distributed numbers ranging from **0** to **1**. Why?
Because starting from uniform random number generators, most of the other distributions of interest can be derived using specific algorithms.
It is intuitive that at a certain point the sequence of numbers will repeat itself.
It is crucial that the *period of a random sequence* (the number of extractions after which the sequence will repeat itself) should be as large as possible and, anyway, larger than the number of random numbers required in a specific application.

Uniform random number generators

The Monte Carlo method is a numerical technique for …
… calculating probabilities and related quantities by using sequences of random numbers.

The most widely used computer-based random number generators are conveniently written …
.. in order to produce sequences of uniformly distributed numbers ranging from **0** to **1**. Why?
Because starting from uniform random number generators, most of the other distributions of interest can be derived using specific algorithms.
It is intuitive that at a certain point the sequence of numbers will repeat itself.
It is crucial that the *period of a random sequence* (the number of extractions after which the sequence will repeat itself) should be as large as possible and, anyway, larger than the number of random numbers required in a specific application.

One example is the function **`lrand48()`** (a standard of C programming language) - defined by the algorithm discussed in next slide - that has a period of $\sim 2^{48}$. The sequences for a *given* but *generic initial* (*) value x_0 (called ***seed***) are distributed uniformly in the interval $[0, 2^{31}[$. The obtained sequences of random bits can be used to return 48-bits integer values, or can be mapped (**) into sequences of floating-point numbers uniformly distributed in [0, 1[as implemented in **`drand48()`**.

(*) Note that by choosing different initial seeds, different sequences are produced (reproducibility comes with the same seed).
Thus one can repeat a computer-simulated experiment using different random sequences, each time changing the initial seed & obtaining different results, in order to simulate the statistical fluctuations occurring in reality when repeating an experiment.

(**) Remapping from [0, 1[to [a, b[can be simply obtained by the linear transformation $x' = a + x(b-a)$

Example: `lrand48()` uses the multiplicative linear congruential algorithm (MLCG)

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int lrand48(void);
```

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The lrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

(Multiplicative Linear Congruential Generator)

The initial values of X, a, and c are:

```
X(0) = 1
a    = 5deece66d  (base 16)
c    = b          (base 16)
```

Note: with a and c constant and the value given to the seed the entire sequence is strictly determined!

Source: https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-lrand48-pseudo-random-number-generator

Example: `lrand48()` uses the multiplicative linear congruential algorithm (MLCG)

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int lrand48(void);
```

Returned value

lrand48() transforms the generated 48-bit value, X(n+1), to a nonnegative, long integer value on the interval [0,2**31) and returns this transformed value.

Note: with a and c constant and the value given to the seed the entire sequence is strictly determined!

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The lrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)      n>=0
```

(Multiplicative Linear Congruential Generator)

The initial values of X, a, and c are:

```
X(0) = 1
a    = 5deece66d  (base 16)
c    = b          (base 16)
```

Source: https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-lrand48-pseudo-random-number-generator

Example: `lrand48()` uses the multiplicative linear congruential algorithm (MLCG)

Format

```
#define _XOPEN_SOURCE
#include <stdlib.h>

long int lrand48(void);
```

Returned value

lrand48() transforms the generated 48-bit value, X(n+1), to a nonnegative, long integer value on the interval [0,2**31) and returns this transformed value.

The modulo (mod) operator means that aXmod(m) takes the remainder of aX divided by m, where the multiplier a and the modulus m are integer;
Ex: 27mod5=2

Note: with a and c constant and the value given to the seed the entire sequence is strictly determined!

General description

The drand48(), erand48(), jrand48(), lrand48(), mrand48() and nrand48() functions generate uniformly distributed pseudo-random numbers using a linear congruential algorithm and 48-bit integer arithmetic.

The functions drand48() and erand48() return nonnegative, double-precision, floating-point values, uniformly distributed over the interval [0.0,1.0).

The functions lrand48() and nrand48() return nonnegative, long integers, uniformly distributed over the interval [0,2**31).

The functions mrand48() and jrand48() return signed long integers, uniformly distributed over the interval [-2**31,2**31).

The lrand48() function generates the next 48-bit integer value in a sequence of 48-bit integer values, X(i), according to the linear congruential formula:

```
X(n+1) = (aX(n) + c)mod(2**48)     n>=0
```

(Multiplicative Linear Congruential Generator)

The initial values of X, a, and c are:

```
X(0) = 1
a    = 5deece66d  (base 16)
c    = b          (base 16)
```

Source: https://www.ibm.com/docs/en/zos/2.1.0?topic=functions-lrand48-pseudo-random-number-generator

The transformation method

Nonuniformly distributed random numbers can be generated starting from a uniform random number generators using various algorithms.

Given a sequence of random numbers $r_1, r_{2, \dots}$ uniformly distributed in $[0\ ,1]$, the next step is to determine a sequence $x_1, x_{2, \dots}$ distributed according to the p.d.f. $f(x)$ in which one is interested.

In the **transformation method** this is accomplished by finding a suitable function $x(r)$ which directly yields the desired sequence when evaluated with the uniformly generated r values; in other words, ... the task consists in finding a function $x(r)$ that is distributed according to a specified $f(x)$ given that r follows a uniform distribution in $[0\ ,1]$.

The transformation method

Nonuniformly distributed random numbers can be generated starting from a uniform random number generators using various algorithms.

Given a sequence of random numbers $r_1, r_{2,\dots}$ uniformly distributed in $[0,1]$, the next step is to determine a sequence $x_1, x_{2,\dots}$ distributed according to the p.d.f. $f(x)$ in which one is interested.

In the **transformation method** this is accomplished by finding a suitable function $x(r)$ which directly yields the desired sequence when evaluated with the uniformly generated r values; in other words, ... the task consists in finding a function $x(r)$ that is distributed according to a specified $f(x)$ given that r follows a uniform distribution in $[0,1]$.

We can start from the cumulative distribution that can be built: $F(x) = \int_{-\infty}^{X} f(x')dx'$

By inverting the cumulative distribution (*) it is possible to demonstrate that ...

... extracting a random number r uniformly distributed in $[0,1[$ ... the transformed variable $x = F^{-1}(r)$ is distributed according to $f(x)$

(*)Note: this method is useful when the cumulative $F(x)$ can be easily computed and inverted either analitically or numerically.

The transformation method

Nonuniformly distributed random numbers can be generated starting from a uniform random number generators using various algorithms.

Given a sequence of random numbers $r_1, r_{2, \dots}$ uniformly distributed in $[0,1]$, the next step is to determine a sequence $x_1, x_{2, \dots}$ distributed according to the p.d.f. $f(x)$ in which one is interested.

In the **transformation method** this is accomplished by finding a suitable function $x(r)$ which directly yields the desired sequence when evaluated with the uniformly generated r values; in other words, ... the task consists in finding a function $x(r)$ that is distributed according to a specified $f(x)$ given that r follows a uniform distribution in $[0,1]$.

We can start from the cumulative distribution that can be built: $F(x) = \int_{-\infty}^{X} f(x')dx'$

By inverting the cumulative distribution (*) it is possible to demonstrate that ...

... extracting a random number r uniformly distributed in $[0,1[$ ... the transformed variable $x = F^{-1}(r)$ is distributed according to $f(x)$

It is straightforward to demonstrate that x follows the desired p.d.f.:

$$r = F(x) \Rightarrow dr = \frac{dF}{dx}dx = f(x)dx$$

Introducing the differential probability dP: $\frac{dP}{dx} = f(x)\frac{dP}{dr}$

Since r is uniformly distributed: $\frac{dP}{dr} = 1$

$$\frac{dP}{dx} = f(x)$$

(*)Note: this method is useful when the cumulative $F(x)$ can be easily computed and inverted either analitically or numerically.

Hit-or-Miss (or acceptance-rejection method) Monte Carlo

In case the cumulative distribution of a p.d.f. cannot be easily computed and inverted, neither analytically nor numerically, other methods allow generating random numbers according to the desired PDF with reasonably good CPU performances.

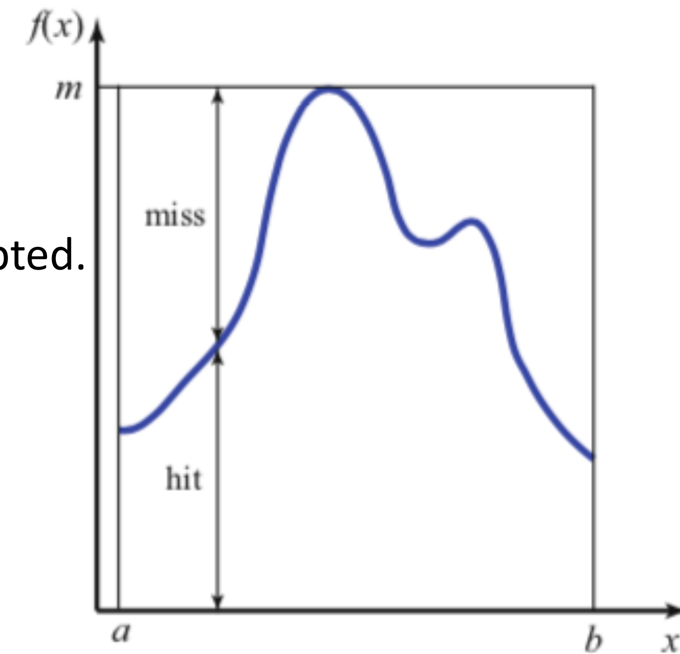
A rather general-purpose and simple random number generator is the **hit-or-miss Monte Carlo**.

It assumes a p.d.f. $f(x)$ defined in an $x \in [a, b[$ interval , not necessarily normalized (i.e. in general $\int_a^b f(x)dx \neq 1$)

The maximum value (m) of $f(x)$ must be known! The method proceeds as follows:

1) a uniform random number x is extracted in the interval $[a, b[$ and $f(x)$ is computed;

2) a random number r is extracted uniformly in the interval $[0, m[$;
if $\mathrm{r} > f$ ("miss") the extraction of x is repeated until $\mathrm{r} < f$ ("hit") and x is extracted/accepted.



Hit-or-Miss (or acceptance-rejection method) Monte Carlo

In case the cumulative distribution of a p.d.f. cannot be easily computed and inverted, neither analytically nor numerically, other methods allow generating random numbers according to the desired PDF with reasonably good CPU performances.

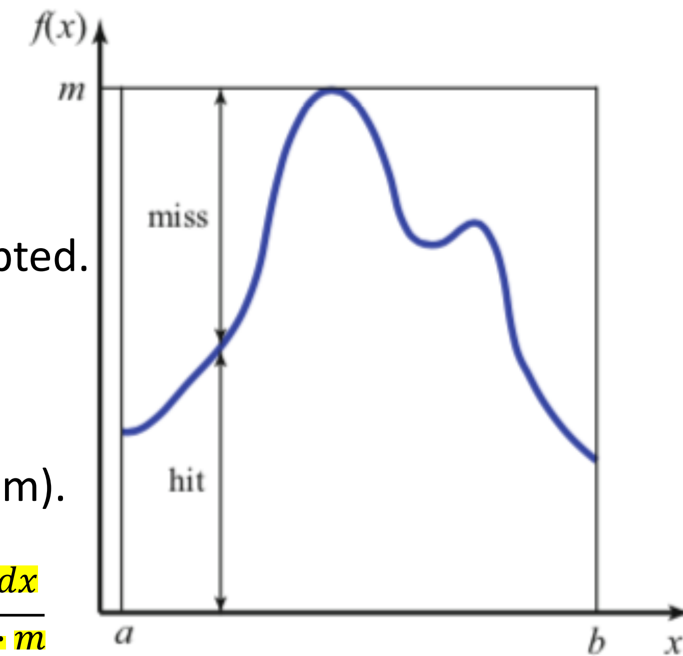
A rather general-purpose and simple random number generator is the **hit-or-miss Monte Carlo**.

It assumes a p.d.f. $f(x)$ defined in an $x \in [a, b[$ interval , not necessarily normalized (i.e. in general $\int_a^b f(x)dx \neq 1$)

The maximum value (m) of $f(x)$ must be known! The method proceeds as follows:

1) a uniform random number x is extracted in the interval $[a, b[$ and $f(x)$ is computed;

2) a random number r is extracted uniformly in the interval $[0, m[$;
   if $\mathrm{r} > f$ ("miss") the extraction of x is repeated until $\mathrm{r} < f$ ("hit") and x is extracted/accepted.



The probability distribution of the accepted values of x is equal to the initial p.d.f. $f(x)$, up to a normalization factor, by construction. Therefore, the method rejects a fraction of extractions equal to the ratio of the area under the curve to the area of the rectangle (a,b,0,m). In other words, this method has an *efficiency* (i.e. the fraction of accepted values):

$$\varepsilon = \frac{\int_a^b f(x)dx}{(b-a)\cdot m}$$

Since ε<1 there is clearly a non optimal use of computing power;
in particular if the shape of $f(x)$ is very peaked the method will behave very slowly.

Note: this method can also be applied to multidimentional cases with no conceptual difference [with $\vec{x}$ and $f(\vec{x})$].

Example with acceptance-rejection method

In this case (illustrated by Cowan) we consider the variable $x \in [-1,+1]$, following the p.d.f. $f(x) = \frac{3}{8}(1+x^2)$
It's the 1st random number to be generated and uniformly distributed in its interval: $x = x_{min} + r_1 \cdot (x_{max} - x_{min})$, $r_1 \in [0,1]$

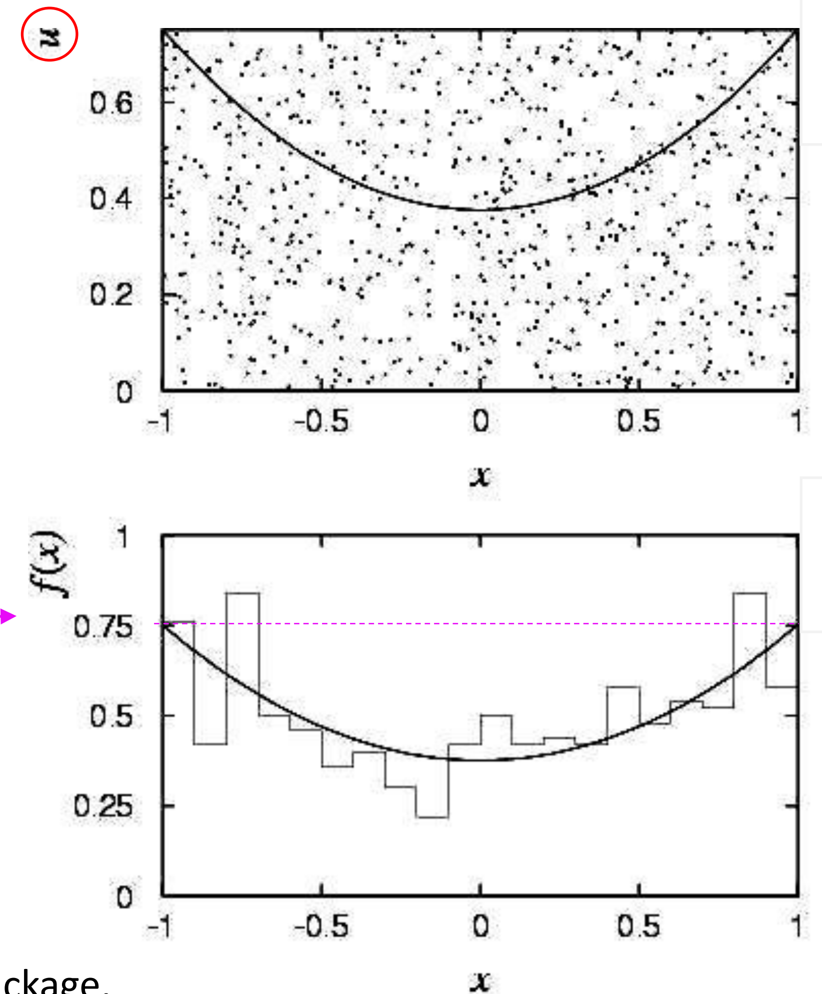
At $x = \pm 1$ the p.d.f. reaches its maximum value of $f_{max} = \frac{3}{4}$

The 2nd **independent** random number $\boldsymbol{u}$ will be uniformly distributed between 0 & f_{max} , namely $\boldsymbol{u} = r_2 \cdot f_{max}$ (with $r_1 \in [0,1]$)

Scatter plot of the random numbers u and x generated according to the given algorithm.

The x values of the points that lie below the curve are accepted [i.e. when $u < f(x)$].

This plot shows a normalized histogram built from the accepted points.



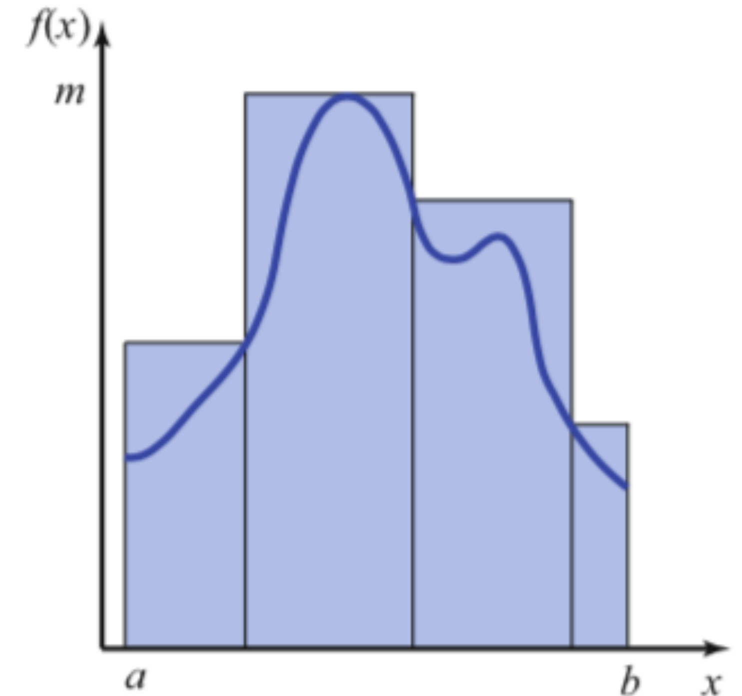
The efficiency is: $\varepsilon = \frac{\int_{-1}^{+1} f(x)dx}{(1-(-1)) \cdot f_{max}} = \frac{1}{2 \cdot 3/4} = \frac{2}{3}$

Note: in the SDAL (lab) course - next year - we will generate observables defined in a generic interval and distributed according to a generic distribution with the `RooFit/Root` package.

Variations of the Hit-or-Miss Monte Carlo

The hit-or-miss algorithm can be adapted to improve the efficiency by identifying, in a preliminary stage of the algorithm, a partition of the interval $[a, b[$ such that, in each subinterval, the function $f(x)$ has a smaller variation than in the overall range.

In the so-called ***importance sampling*** approach, the modified algorithm proceeds considering a different maximum in each subinterval, where the latter is randomly chosen with a probability proportional to its area.



A possible variation of this method is to use, instead of the aforementioned partition, an 'envelope' for the function $f(x)$, that is another function $g(x)$ so that $g(x) > f(x)$ $\forall x \in [a, b[$ and for which a convenient method to extract x according to the $g(x)$ is known. As a result of a properly choice of the envelope and of the partition … the efficiency can improve a lot!

Numerical integration with Monte Carlo methods

The Monte Carlo method can be applied whenever the solution to a problem can be related to a parameter of a probability distribution. This could be either an explicit parameter in a p.d.f., or the integral of the distribution over some region.
A sequence of Monte Carlo generated values is used to evaluate an estimator for the parameter (or **integral**), just as is done with real data.
An important feature of properly constructed estimators is that their statistical accuracy improves as the number of values $\boldsymbol{n}$ in the data sample (from Monte Carlo or otherwise) increases. It is possible to show that under fairly general conditions …
… the standard deviation of an estimator is inversely proportional to $\sqrt{\boldsymbol{n}}$.
The Monte Carlo method thus represents a numerical integration technique for which the accuracy increases as $1/\sqrt{\boldsymbol{n}}$.

This scaling behaviour with the number of generated values can be compared to the number of points necessary to compute an integral using the *trapezoidal rule* for which the accuracy improves as $1/\boldsymbol{n}^2$, namely much faster than by Monte Carlo.
However, for an integral of dimension $\boldsymbol{d}$, this accuracy changes to $1/\boldsymbol{n}^{2/d}$,
… whereas **for Monte Carlo integration one has** $1/\sqrt{\boldsymbol{n}}$ ***for any dimension*** [*].
Thus: for $\boldsymbol{d} > 4$ the dependence of the accuracy on n is better for the Monte Carlo method!

For other integration methods, such as *Gaussian quadrature*, a somewhat better rate of convergence can be achieved than for the trapezoidal rule. For a large enough number of dimensions, however, the Monte Carlo method will always be superior.

[*] this is argued @ next slide

Hit-or-miss method : how numerical integration accuracy depends on the # of extractions

As discussed in the previous slide… Monte Carlo methods are often used as numerical techniques in order to compute integrals.

The *hit-or-miss* method estimates the integral $\int_a^b f(x)dx$ from the fraction of accepted hits $\hat{n}$ over the total # of extractions N:

$$I = \int_a^b f(x)dx \cong \hat{I} = (b-a)\cdot\frac{\hat{n}}{N} \propto \frac{\hat{n}}{N}$$

With this approach, $\hat{n}$ follows a binomial distribution. Thus, remember that the uncertanty on $\hat{\varepsilon} = \hat{n}/N$ is $\sigma_{\hat{\varepsilon}} \cong \sqrt{\frac{\hat{\varepsilon}(1-\hat{\varepsilon})}{N}}$
Therefore, if $\hat{n}$ is not too close to neither 0 nor N (extreme cases for which $\sigma_{\hat{\varepsilon}} \cong 0$),
the uncertainty on integral is:

$$\sigma_{\hat{I}} \cong (b-a)\cdot\sqrt{\frac{\hat{I}(1-\hat{I})}{N}}$$

The uncertainty on $\hat{I}$ decreases with $\sqrt{N}$!

Note: this result is true also if the hit-or- miss method is applied to a multidimensional integration, regardless of the number of dimensions of the problem!

Monte Carlo method for the simulation of experimental data - I

The Monte Carlo method is often used to simulate experimental data.

In HEP this is typically done in two stages: 1) **event generation** & 2) **detector simulation**.

Consider, for example, an experiment in which an incoming particle such as an electron scatters off a target and is then detected. Suppose there exists a theory that predicts the probability for an event to occur as a function of the scattering angle (i.e. the differential cross section).

First one constructs a Monte Carlo program to generate values of the scattering angles, and thus the momentum vectors, of the final state particles. Such a program is called an **event generator**.

In HEP, event generators are available to describe a wide variety of particle reactions. The output of the event generator, i.e. the momentum vectors of the generated particles, is then used as input for a detector simulation program.

Monte Carlo method for the simulation of experimental data - II

The Monte Carlo method is often used to simulate experimental data.

In HEP this is typically done in two stages: 1) **event generation** & 2) **detector simulation**.

Consider, for example, an experiment in which an incoming particle such as an electron scatters off a target and is then detected. Suppose there exists a theory that predicts the probability for an event to occur as a function of the scattering angle (i.e. the differential cross section).

First one constructs a Monte Carlo program to generate values of the scattering angles, and thus the momentum vectors, of the final state particles. Such a program is called an **event generator**.

In HEP, event generators are available to describe a wide variety of particle reactions. The output of the event generator, i.e. the momentum vectors of the generated particles, is then used as input for a detector simulation program.

Since the response of a detector to the passage of the scattered particles also involves random processes such as the production of ionization, multiple Coulomb scattering, etc., the detector simulation program is also implemented using the Monte Carlo method.

Programming packages such as **GEANT4** **(*)** can be used to describe complicated detector configurations, and experimental collaborations typically spend considerable effort in achieving as complete a modelling of the detector as possible. This is especially important in order to optimize the detector's design for investigating certain physical processes before investing time and money in constructing the apparatus.

(*) https://geant4.web.cern.ch/

Markov Chain Monte Carlo

The Monte Carlo methods considered so far are based on **sequences of uncorrelated pseudorandom numbers** that follow a given probability distribution. There are classes of algorithms that sample more efficiently some probability distributions by producing sequences of ***correlated*** **pseudorandom numbers**, i.e. each entry in the sequence depends on previous entries.

For instance **a sequence of random variables is a Markov Chain if** … the probability of a variable "n+1" depends ONLY on the previously ("n") extracted value and NOT on all the previous values (from "0" to "n"), characteristic that corresponds to a "loss of memory". The Markov Chain is said to be ***homogeneous*** if the probability of a variable - correlated to its previous one in the sequence - does not depend on the position in the chain.

One example of Markov Chain is the Metropolis-Hastings algorithm (for details see: L.Lista, section 4.8, p.93).