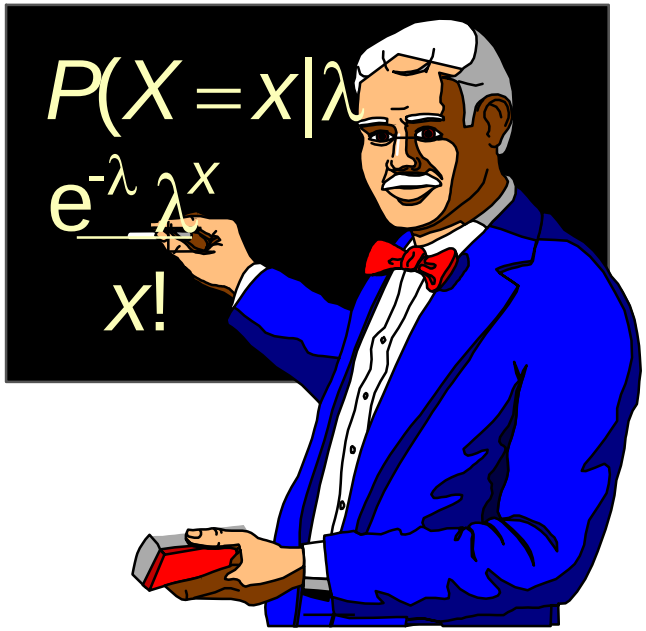
Phys 443
Computational Physics

Monte Carlo Simulation



Simulation

- Simulation is one of the most widely used decision modeling techniques

Pro:

- Flexibility
- Can handle large and complex systems
- Can answer “what-if” questions
- Does not interfere with the real system
- Allows study of interaction among variables

Contra:

- Can be time consuming
- Does not generate optimal solutions

Monte Carlo Simulation

- Can be used with variables that are probabilistic

Steps:

- **Determine the probability distribution** for each random variable
- Use random numbers to generate random values
- Repeat for some number of replications

Monte Carlo Method

"Monte Carlo" methods are a broad set of techniques for calculating probabilities and related quantities using sequences of random numbers. "

Developed by S. Ulam, J. von Neumann, and N. Metropolis at LANL in 1946 to model neutron diffusion in radiation shields.

MC simulation

Method was called “Monte Carlo” after the Casino de Monte-Carlo in Monaco.

Metropolis coined the name “Monte Carlo”, from its gambling Casino.

Monte-Carlo, Monaco

MC Techniques

- General idea is: instead of performing long complex calculations, perform large number of "experiments" using random number generation and see what happens.
- Simulate physical systems with models of noise and uncertainty
- Perform calculations that cannot be done analytically e.g. function minimization, or many high-dimensional integrals

Purpose of Monte Carlo Programs

- Monte Carlo programs are used not only for particle physics, but also biology, medicine, meteorology
- Purpose we use in particle/nuclear physics
  - Simulation of physics events which based on random processes/probabilities
  - Numerical implementation of a cross section calculator (calculation of complicated integrals)
  - Simulation of detector response to particles (which again reduces to simulation of physics processes using random numbers)
  - Optimization of detector design, computation of detector resolution, efficiencies, acceptances

- The basic form of random number generator produces a sequence of numbers with a uniform probability in [0,1]

Earliest MC calculation

1777

Compte de Buffon: If a needle of length L is thrown at random onto a plane ruled with straight lines a distance $d(d > L)$ apart, then the probability P of the needle intersecting one of those lines is $P = 2L/\pi d$.

1930s

First significant scientific application of MC: Enrico Fermi used it for *neutron transport* in fissile material.

1940s

1940s Monte Carlo named by Nicholas Metropolis and Stanislaw Ulam

Earliest MC calculation

1953

Algorithm for sampling any probability density

Metropolis, Rosenbluth,and Teller

(generalized by Hastings in 1970)

1962, 1974

First QMC calculations, Kalos, Levesque, Verlet.

1965

First VMC calculations (of liquid He), Bill McMillan.

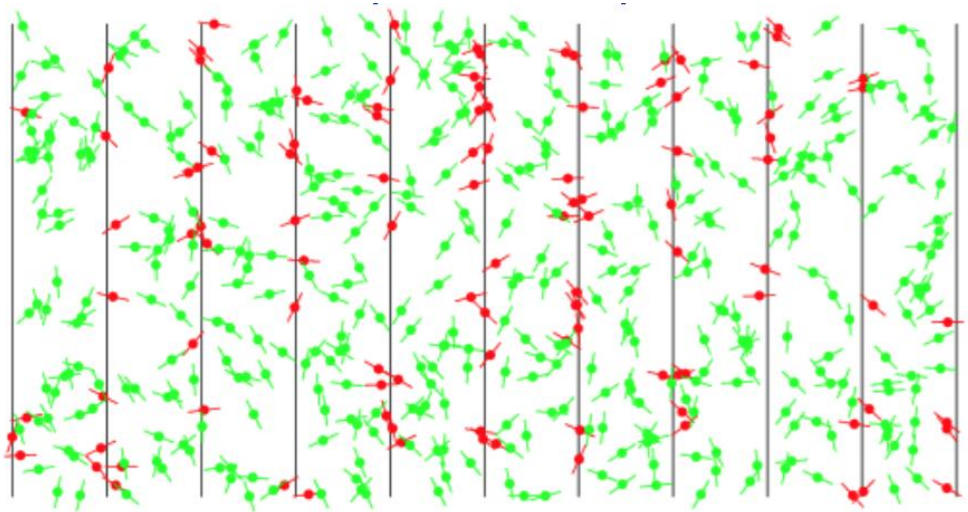
Variational Monte Carlo

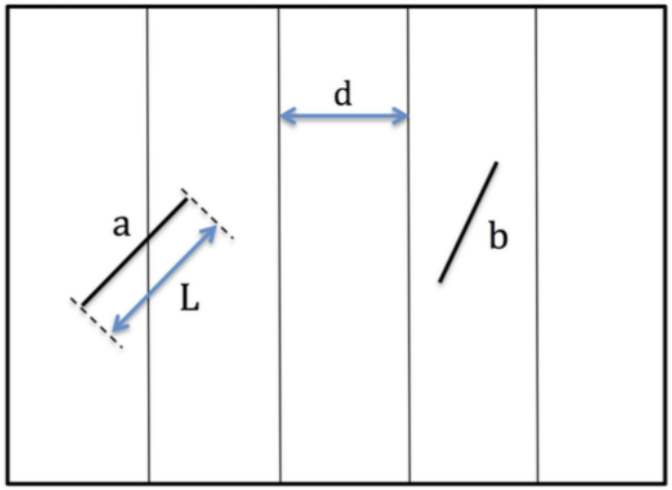
Earliest MC calculation

- Buffon's needle is a very early example of applying MC methods to calculate π (1777).

- Buffon's Algorithm is the following:
  - Lay out a pattern of parallel lines on floor, each separated by a distance *'d'*.
  - Throw a needle of , length 'd' randomly onto the pattern, repeatedly n times
    - If needle crosses a strip boundary ==> hit
    - If needle is between boundaries==> miss
  - Estimate π=2*n/ # of hits

Earliest MC calculation





Earliest MC calculation

- Buffon's needle: Why does it work?
  - It is based on the fact that hit-probability $P(hit)=2/\pi$
- Let angle between needle and perpendicular to line be θ
- Projection of needle onto perpendicular is $=d^*|\cos\theta|$
- Probability for hit = ratio of $d^*|\cos\theta|/d$
- All angles are equal

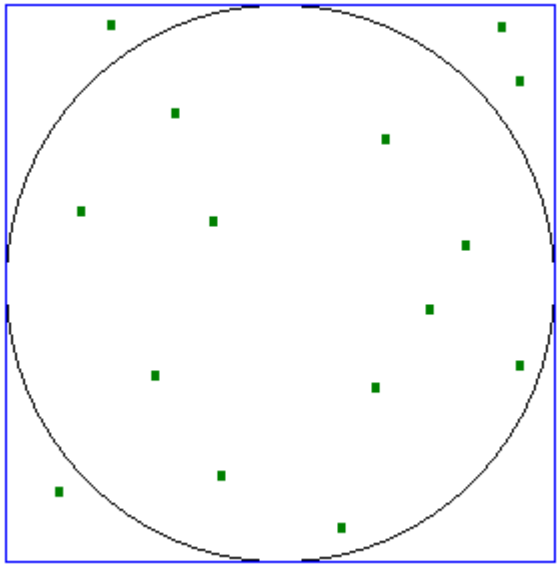
$$<|\cos\theta|>=\frac{1}{\pi/2}\int_0^{\pi/2}\cos\theta\cdot d\theta$$

Earliest MC calculation

- Buffon's needle: What is accuracy?
- Number of hits follows binominal distribution with expectation of n_{nits} follows a binomial distribution

  $E[n_{nits}]=np$ for n trials
- Probability of hit $p=2/\pi$
- **Variance $V[n_{nits}]=np(1-p)=\sigma^2[n_{nits}]$**
- **$\Rightarrow \sigma^2(2/\pi) = p(1-p)/n \Rightarrow \sigma(\pi)=2.37/\sqrt{n}$**
- **Estimate uncertainty $\sigma(\pi)$**
  - **N=100 $\Rightarrow \sigma(p)=0.237$**
  - **N=10000 $\Rightarrow \sigma(p)=0.0237$**
  - **N=1000000 $\Rightarrow \sigma(p)=0.00237$**
- **Hit or miss MC is not efficient ! !**

Earliest MC calculation



Ratio of Random points inside circle vs. outside circle = $\frac{\pi r^2}{2^2} = \frac{\pi}{4}$

≈ 0.7854

Buffon’s needle

```
'''
Buffon's Needle Experiment
n = number of throws
r = number of runs
a = length of needle
b = distance between cracks
theta = angle needle makes to crack
rcenter = center of needles on floor
0  < theta < pi/2
0 < xcenter < b/2

nhits <===  number of hits of needle centered at x, with
orientation theta
nhits = 1 if x < a/2 and abs(theta) < arcos(x/(a/2))
      = 0 otherwise
'''
```

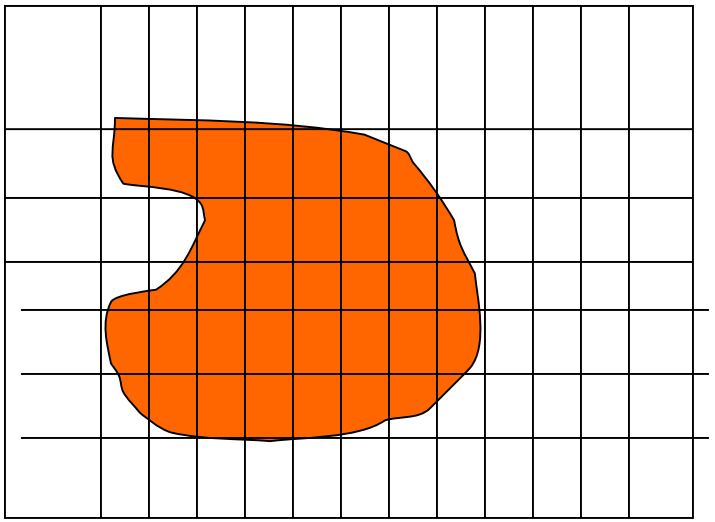
Buffon's needle

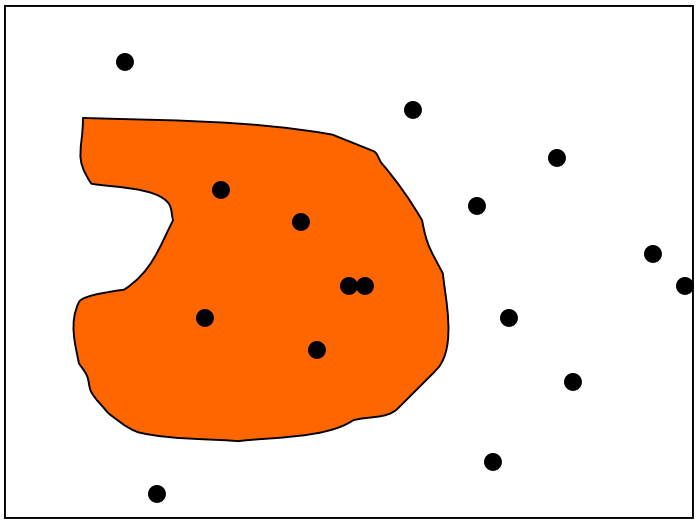
```
import random
import math
def buffon(n,r,a,b):
    data=[]
    print ('Buffon Needle Experiment ')
    print ('estimate of pi')
    for jj in range(r):
        nhits = 0
        for ii in range(n):
            xcent = random.uniform(0,b/2.0)
            theta = random.uniform(0,math.pi/2)
            xtip  = xcent - (a/2.0)*math.cos(theta) #use of cosine not
historically accurate
            if xtip < 0 :
                nhits += 1

        d = b*nhits
        print(jj,nhits, c/d)
        data.append([jj,nhits])
    return data
r=5
n=4000
a = 2  #needle 2 unit length
b = 2  #cracks 2 unit spacing
hits= buffon(n,r,a,b)
```

MC Techniques

- Cover the figure by a grid, calculate the number of grid cells which are inside and this gives you the area

- Shoot at random at the figure. Count the bullets that hit it. The area of then figure is
- S=(Nhit/Ntotal)*S(rectangle)





How to generate random numbers using computer

- The random numbers should be independent.
- The method should be very fast and not require a large amount of computer memory.

How to generate random numbers using computer

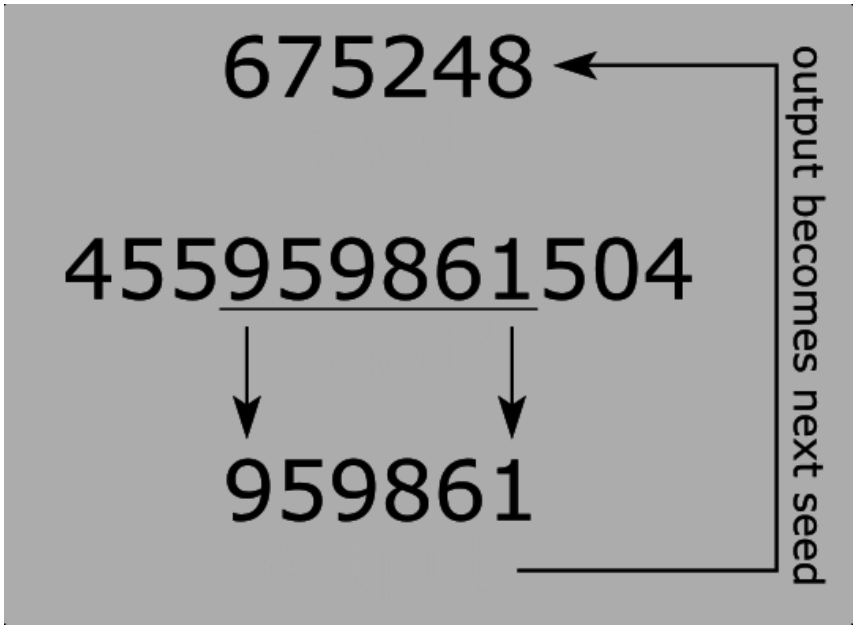
- In principle, the best way to obtain a series of random numbers $x_1, x_2 ... x_n$ is to use some process in nature
  - Throw a coin or dice or needle..
  - Take the decay time of radioactive nuclei
- This is of course is not very efficient
- Therefore computer algorithms have been developed which generate pseudo-random numbers which are "uniformly" distributed in [0,1]

Generating Random Numbers

- The Monte Carlo Method depends upon the generation of random numbers with a computer.
- But a computer is (supposed to be) a deterministic device: given a fixed input and starting conditions, we should always get the same output.
- So we are actually talking about generating pseudorandom numbers.

Arithmetical approach

- Arithmetical approach to producing a sequence of "random" numbers:
  - Start with a "*seed*" value containing n digits
  - Square the number to get a new value with 2n digits; if the result has less than 2n digits, pad it with leading zeros
  - Take the central n digits as the output
  - Use the output as the next "seed" value



Pseudo-Random Number Generators (RNGs)

- Uniform distribution of pseudo-random numbers

$$x_{n+1} = (ax_n + c) \bmod m$$

can be generated by this linear congruential generator. Here a , b and m are carefully chosen fixed integers, a is the multiplier, b is the increment and m is the modulus. The starting value x_0 is called the seed. The random number in the interval [0, 1] is x_n/m. Hence there are at most m random uniformly distributed numbers:

P (x)= 1 if x ϵ [0,1]
P (x)= 0 if x $\notin$ [0,1]

Random number generator

- Example Sequences:

m=10, a=2, c=1	m=10, a=1, c=7
1	1
3 (2*1 + 1 % 10)	8 (1*1 + 7 % 10)
7 (2*3 + 1 % 10)	5 (1*8 + 7 % 10)
5 (2*7 + 1 % 10)	2 (1*5 + 7 % 10)
1 (2*5 + 1 % 10)	9 (1*2 + 7 % 10)
3 (2*1 + 1 % 10)	6 (1*9 + 7 % 10)
7 (2*3 + 1 % 10)	3 (1*6 + 7 % 10)
5 (2*7 + 1 % 10)	0 (1*3 + 7 % 10)
…	…

- Examples for numbers: m=2147483399; a=40692 has a period of $2*10^9$

Pseudo-Random Number Generators (RNGs)

- The pseudo-RNG used in the rand() function of the C standard library is a Linear Congruential Generator (LCG)

- The period of the RNG, defined as the longest number of steps before the sequence starts repeating, is at most 2^m.
- Note: if m is an unsigned integer (uint32_t on most systems) then the period will be at most $2^{32} \approx 4x10^9$. (Note: $2^{64} \approx 10^{18}$)
- For many choices of *m*, the period will be much less than 2^m.
- Theorem: the full period of the LCG is achieved iff *c* and *m* are co-prime, *a -1* is divisible by all prime factors of m, and *a -1* is a multiple of 4 if *m* is a multiple of 4

Pseudo-Random Number Generators (RNGs)

- Note that pseudo-RNGs are deterministic. If you always use the same x_0, a value known as the seed, you always get the same sequence.
- The choice of seed can affect the performance of the LCG; i.e., a poor choice could lead to a period <<m.
- Determinism is great for debugging, but if you generate the same numbers over and over you aren't getting a pseudo-random sequence
- Common mistake 1: accidentally hardcoding the seed into your simulation code
- Common mistake 2: failing to save the seed you used, which makes it hard to regenerate the sequence later for checks

Randomness Tests

- There are a number of different randomness tests that can be used to evaluate a sequence of random numbers.

- Most assume the random numbers generated have a uniform distribution as other distributions are generally transformed from a uniform distribution.

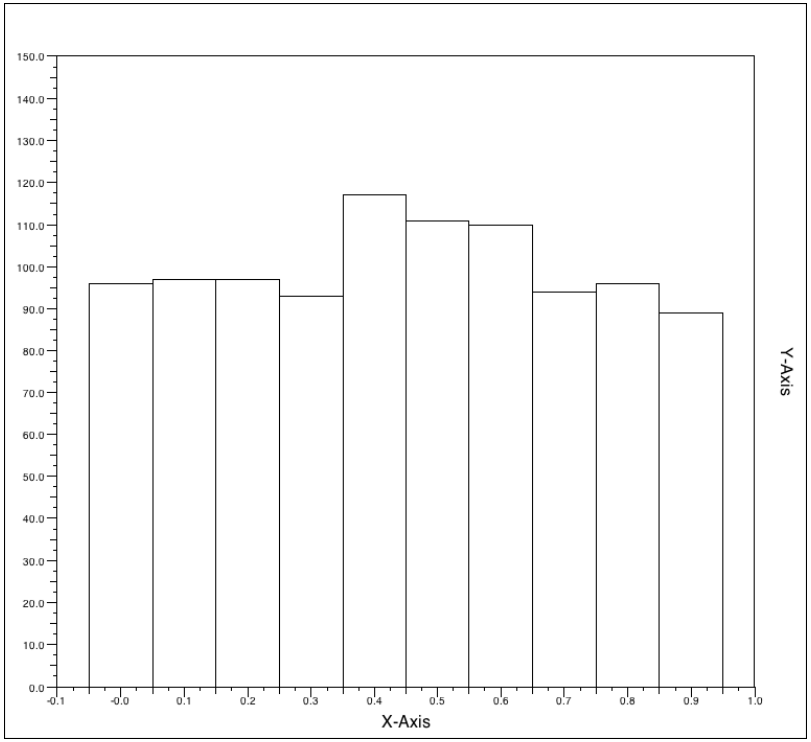
Randomness Tests

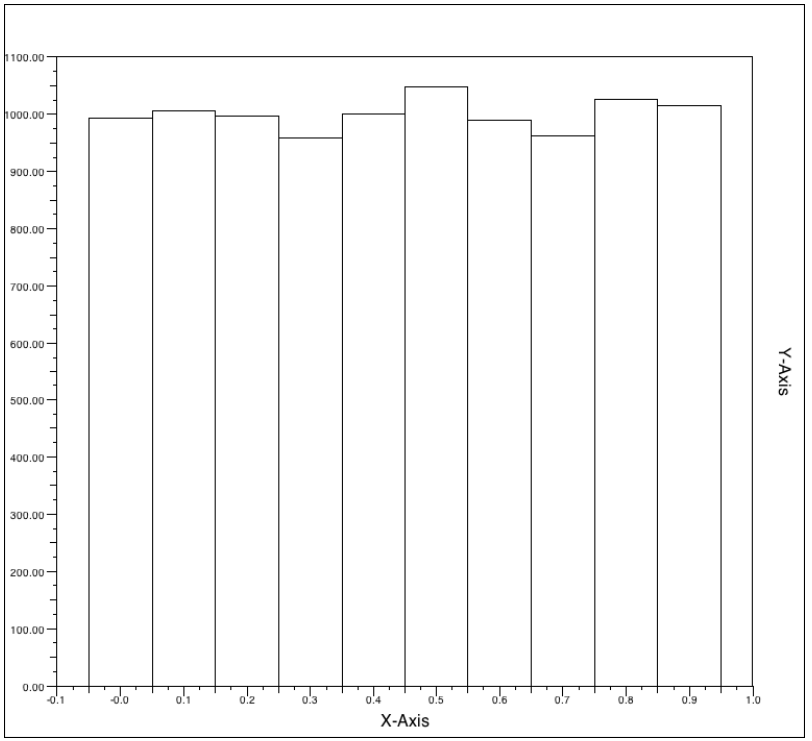
- One simple method to evaluate random number generators is a histogram test.

- Any sequence of uniform random numbers should produce an approximately flat histogram. The larger the number of random numbers tested, the flatter we expect the histogram to be.

Randomness Tests

- 1,000 samples vs 10,000 samples





Random number generators

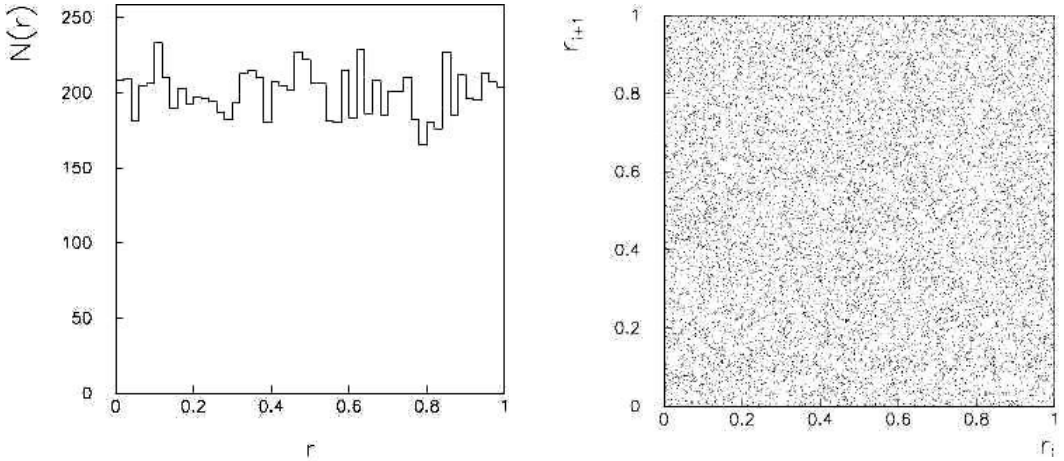
Choose a, m so that the x_i pass various tests of randomness:

uniform distribution in [0, 1],

all values independent (no correlations between pairs),

a = 40692

m = 2147483399

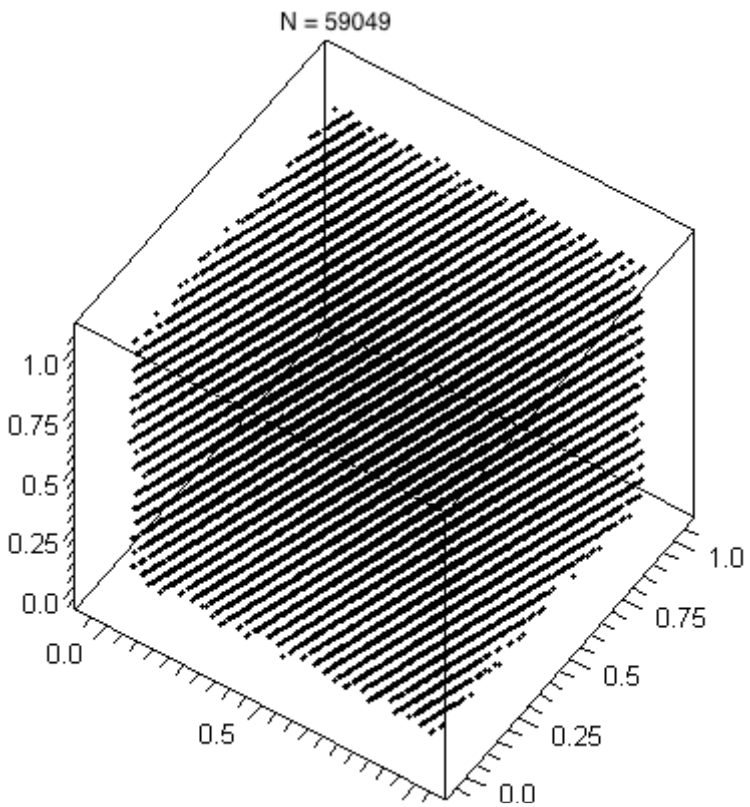


Far better algorithms available, e.g. **RANMAR**, period $\approx 2 \times 10^{43}$.

http://random.mat.sbg.ac.at

Issues with the LCG

- The LCG is fast but will produce subtle long-range correlations between values in the sequence.
- Ex.: if you generate n-dimensional points with the LCG, the points will lie on $(n!m)^{1/n}$ hyperplanes.
- Clearly random numbers shouldn't do that.
- Could this affect your simulation? Maybe. Depends on your application.



Alternatives to the LCG

- A popular RNG currently in use is an algorithm called the Mersenne Twister(MT)
- The MT implementation in Python and C++ (Boost, ROOT) has period 2^{19937} -1 ≈4x 10^{6001}.

Python: Random number generators

```
from random import *
```

`randint(`**`min, max`**`)`

- returns a random integer in range [**min**, **max**] inclusive

`choice(`**`sequence`**`)`

- returns a randomly chosen value from the given sequence
  - the sequence can be a range, a string, ...

Python: Random number generators

```
import random
for i in range(10):
  xn= random.randint(1,101)
  print(xn)
```

```
import random
my_random=[]
for i in range(10):
  xn= random.randint(1,101)
  my_random.append(xn)
  print(my_random)
```

How to obtain a general pdf?

- Two standard methods are
- The transformation method
- The acceptance-rejection method

The Transformation Method

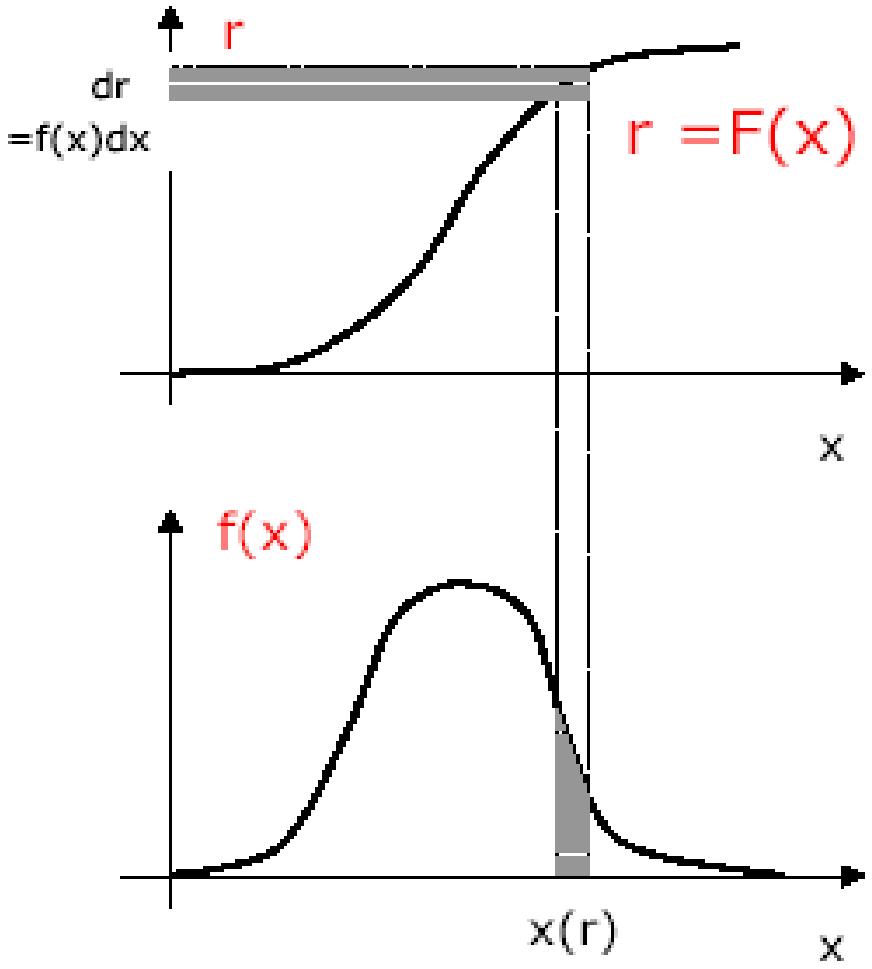
- Job: given a sequence of [0,1] uniform distributed random numbers $r_1, r_2, r_3, \ldots$ (with pdf g(x)=1)
- Find function x(r) such that f(x(r))=G(r) where F and G are cumulative distributions of f(x) and g(r).
- The cumulative function G of the uniform pdf g(x) is G(r)=r

$$F(x) = \int_{-\infty}^{x} f(x') \cdot dx'$$

2. G(r) = F(x(r)) = r
   ➔ and solve to get x(r)

$$\frac{\partial F(x)}{\partial x} = f(x)$$

Produce pdf via transformation



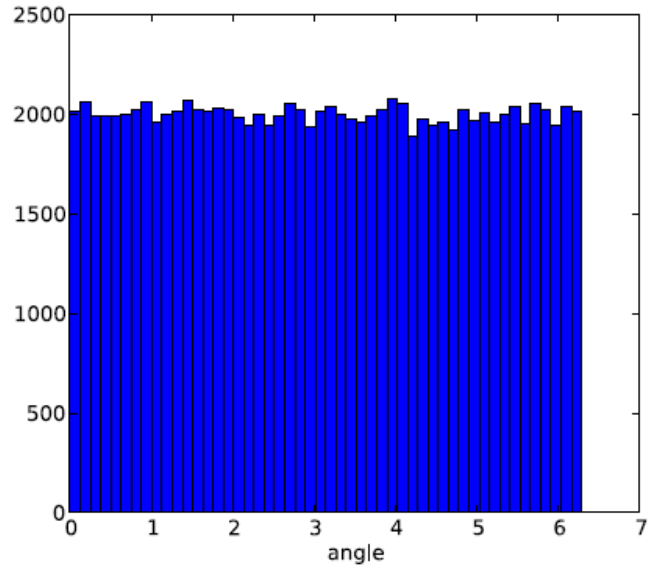
1. Draw r ∈ [0,1] from a uniform pdf
2. Use the relation r

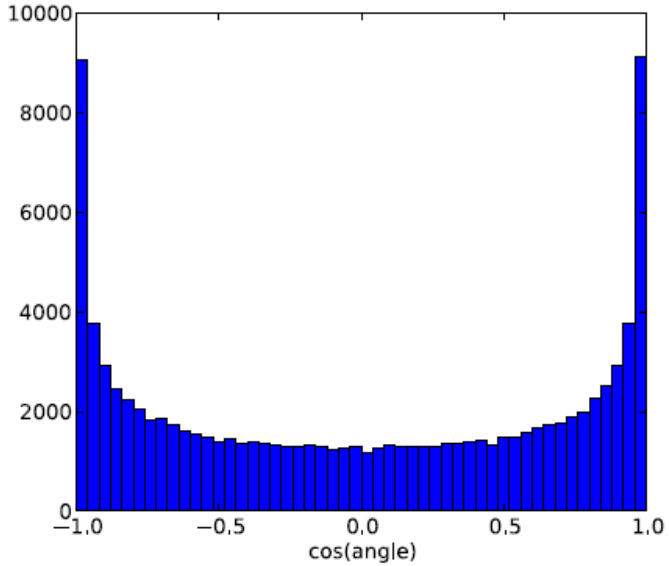
$$r = F(x) = \int_{-\infty}^{x} f(t) \cdot dt$$

3. Invert F(x(r)) to get x as function of r
4. This x is distributed according to pdf f(x)

Produce pdf via transformation

```python
import numpy as np
import matplotlib.pyplot as plt

N_MC=100000  # number of Monte Carlo Experiments
nBins = 50 # number of bins for Histograms

data_x, data_y = [],[]  #lists that will hold x and y

# do experiments
for i in range(N_MC):
    # generate observation for x
    x = np.random.uniform(0,2*np.pi)

    y = np.cos(x)
    data_x.append(x)
    data_y.append(y)

#setup figures
fig = plt.figure(figsize=(13,5))
fig_x = fig.add_subplot(1,2,1)
fig_y = fig.add_subplot(1,2,2)

fig_x.hist(data_x,nBins)
fig_x.set_xlabel('angle')

fig_y.hist(data_y,nBins)
fig_y.set_xlabel('cos(angle)')

plt.show()
```



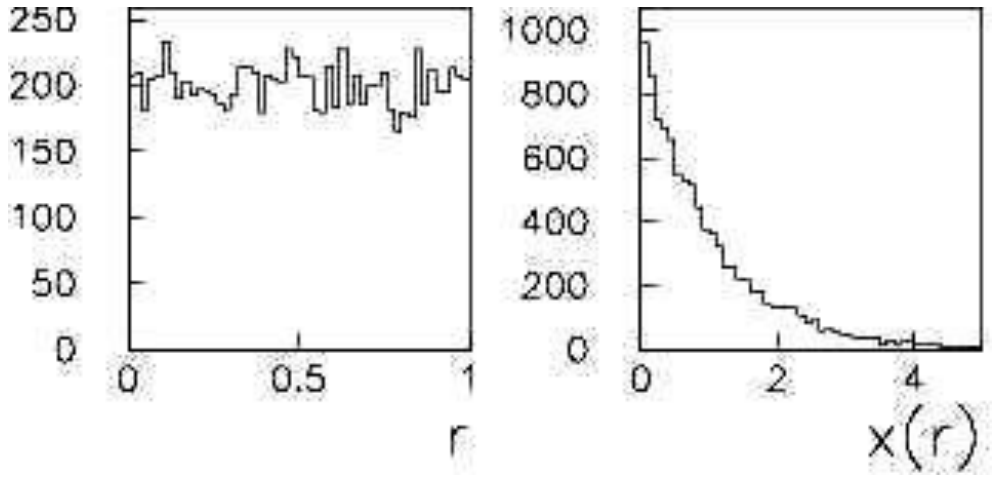


Produce *pdf* via transformation

Exponential pdf: $$f(x;\xi) = \frac{1}{\xi} e^{-x/\xi} \quad (x \geq 0)$$

Set $$\int_0^x \frac{1}{\xi} e^{-x'/\xi}\, dx' = r$$ and solve for x (r).

$$\rightarrow \quad x(r) = -\xi \ln(1-r) \qquad (\, x(r) = -\xi \ln r \quad \text{works too.})$$



Produce pdf via transformation

$$f(y) = ae^{-ay},\ y \in [0, \infty)$$

$$\left|\frac{dx}{dy}\right| = ae^{-ay}, \quad \text{or,} \quad x = e^{-ay}, \quad \text{i.e.,} \quad \boxed{y = \frac{-\ln(x)}{a}}$$

$$f(y) = \frac{y^{-1/2}}{2},\ y \in [0, 1]$$

$$\left|\frac{dx}{dy}\right| = \frac{y^{-1/2}}{2}, \quad \text{or} \quad x = y^{1/2}, \quad \text{i.e.,} \quad \boxed{y = x^2}$$

39

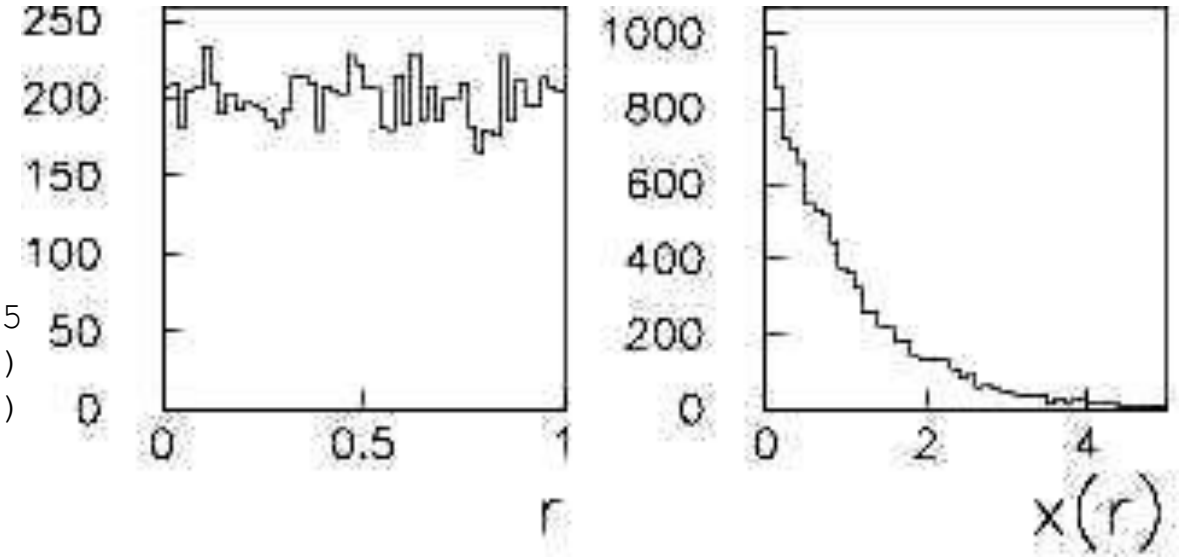
Produce pdf via transformation

$$f(y) = ye^{-y^2/2},\ y \in [0, \infty)$$

$$\left|\frac{dx}{dy}\right| = ye^{-y^2/2}, \quad \text{or,} \quad x = e^{-y^2/2}, \quad \text{i.e.,} \quad \boxed{y = \sqrt{-2\ln(x)}}$$

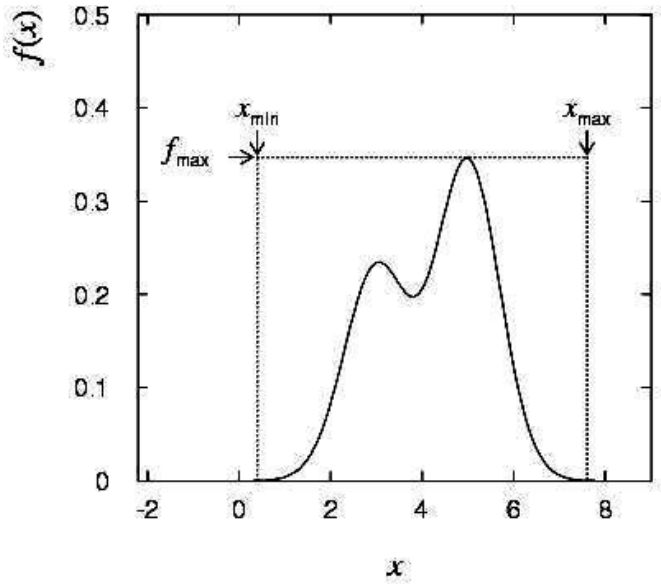
Produce *pdf* via transformation

```
1import numpy as np
2import matplotlib.pyplot as plt
3N_MC = 10000 # number of Monte Carlo Experiments
4nBins = 50 #number of bins for Histograms
5data_x, data_y = [], []  #create list that hold x and y
6#do experiments
7for  i in range(N_MC):
8 # generate MC data
9    x = np.random.uniform()
10  y = -np.log(1-x)
11  data_x.append(x)
12  data_y.append(y)
13#
14#setup figure
15fig=plt.figure(figsize=(13,5
16fig_x=fig.add_subplot(1,2,1)
17fig_y=fig.add_subplot(1,2,2)
18#
19fig_x.hist(data_x,nBins)
20fig_x.set_xlabel('r')
21#
22fig_y.hist(data_y,nBins)
23fig_y.set_ylabel('x(r)')
24#
25plt.show()
```



The acceptance-rejection method

Enclose the pdf in a box:



(1) Generate a random number x, uniform in $[x_{min}, x_{max}]$, i.e.

$$x = x_{\min} + r_1(x_{\max} - x_{\min}) \ , \ r_1 \text{ is uniform in [0,1].}$$

(2) Generate a 2nd independent random number u uniformly distributed between 0 and f_{max}, i.e. $u = r_2 f_{\max}$.

(3) If $u < f(x)$, then accept x. If not, reject x and repeat.

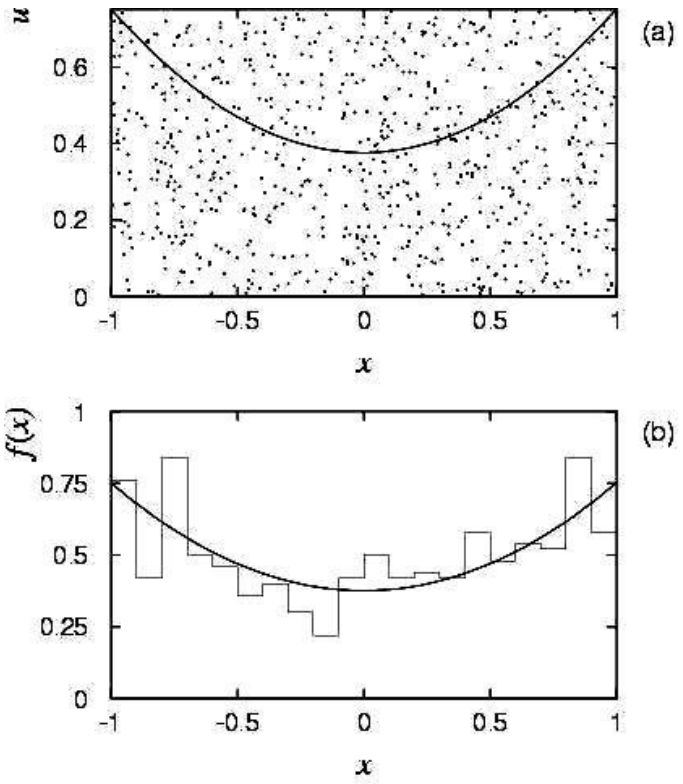
The acceptance-rejection method

At x=±1 p.d.f has a maximum value of f_{max}=3/4

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   xmin=-1
2   xmax=1
3   fmax=0.75
3       N_MC = 10000 # number of Monte Carlo Experiments
4       nBins = 50 #number of bins for Histograms
5   data_x, data_y = [], []  #create list that hold x and y
6    #do experiments
7   for  i in range(N_MC):
8     # generate MC data
9     x1 = np.random.uniform()
10    x2 = np.random.uniform()
11    r=xmin+x1*(xmax-xmin)
12    u= x2*0.75
13    if x<u
14     data_x.append(x1)
15     data_y.append(r)
16   #
17  #setup figure fig=plt.figure(figsize=(13,5))
18  fig_x=fig.add_subplot(1,2,1)
19  fig_y=fig.add_subplot(1,2,2)
20  #
21  fig_x.hist(data_x,nBins)
22  fig_x.set_xlabel('r')
23  #
24  fig_y.hist(data_y,nBins)
25  fig_y.set_xlabel('x(r)')
26  #
27  plt.show()
```

$$f(x) = \frac{3}{8}(1+x^2)$$

$$(-1 \leq x \leq 1)$$

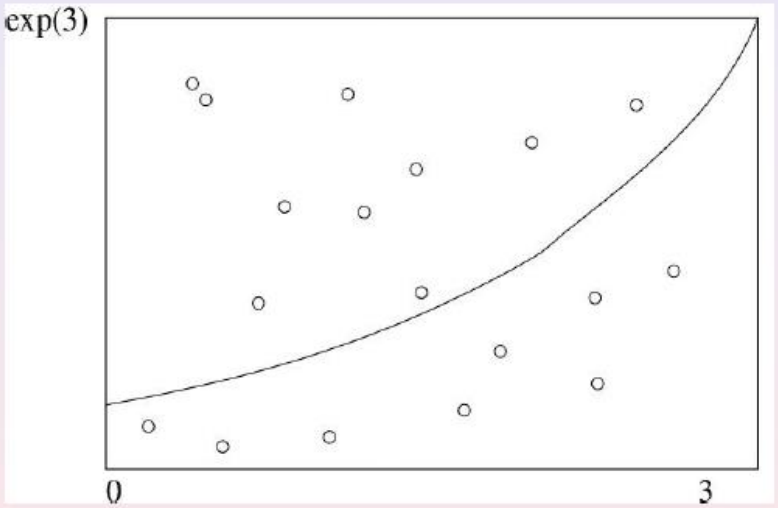


The acceptance-rejection method

```
1  import numpy as np
2  import math
3  import matplotlib.pyplot as plt
4  s=0
5  N_MC = 100000 # number of Monte Carlo Experiments
6  #do experiments
7  for  i in range(N_MC):
8    # generate MC data
9    x= 3.0*np.random.uniform()
10   y = math.exp(3.0)*np.random.uniform()
11   if y<math.exp(x):
12     s=s+1.0
13  #
14 integral= 3.0*math.exp(3.0)*(s/N_MC)
15 print(integral,s,N_MC)
```

$$I = \int_0^3 \exp(x)dx. = 19.085$$

=19.285



Monte Carlo Integration

- Integration of "well-behaved" function: $$I = \int_a^b f(x) \cdot dx$$

1. Useful, simple approximation is (binning in dx): $$I \cong \frac{b-a}{N} \cdot \sum_{i=1}^{N} f(x_i)$$

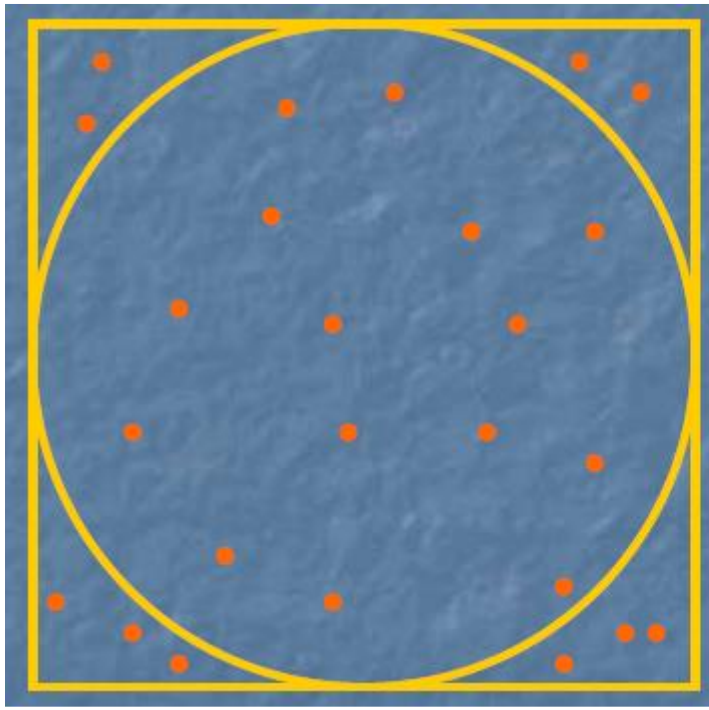
2. More generally, can rewrite (if g(x) known pdf in [a,b]): $$I = \int_a^b \frac{f(x)}{g(x)} \cdot g(x) \cdot dx$$

Since g(x) is a pdf, can calculate an estimator for sample mean (for N trials) , with error σ_I= 1/sqrt(N) :

$$I = E\left[\frac{f(x)}{g(x)}\right] \cong \frac{1}{N} \cdot \sum_{i=1}^{N} \frac{f(x_i)}{g(x_i)}$$

The acceptance-rejection method

A standard example is to use the Monte Carlo method to find the area of a circle and hence determine a value for π .



$$\int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i)$$

$$\pi = \int_{-1}^{1}\int_{-1}^{1} p(x,y)dxdy \approx \frac{4}{N}\sum_{i=1}^{N} p(x_i,y_i)$$

$$p(x,y) = \begin{cases} 1, & x^2+y^2 \le 1 \\ 0, & \text{otherwise} \end{cases}$$

The acceptance-rejection method

Suppose we pick N random points uniformly distributed in a multidimensional volume V: x_1, x_2, x_3,...,x_n. Then the basic theorem of Monte Carlo integration estimates the integral of a function f(x) as:

$$\int f(x)dV \approx \langle f \rangle V \pm V\sqrt{\frac{\langle f^2 \rangle - \langle f \rangle^2}{N}}$$

$$\langle f \rangle = \frac{1}{N}\sum_{i=1}^{N} f(x_i), \quad \langle f^2 \rangle = \frac{1}{N}\sum_{i=1}^{N} f^2(x_i)$$

$$\sigma = \sqrt{\frac{1}{(N_{trials}-1)}\sum_{i=1}^{N_{trials}} (I_i - \bar{I})^2} \Rightarrow$$

$$S_I = \frac{\sigma}{\sqrt{N_{trials}}}$$

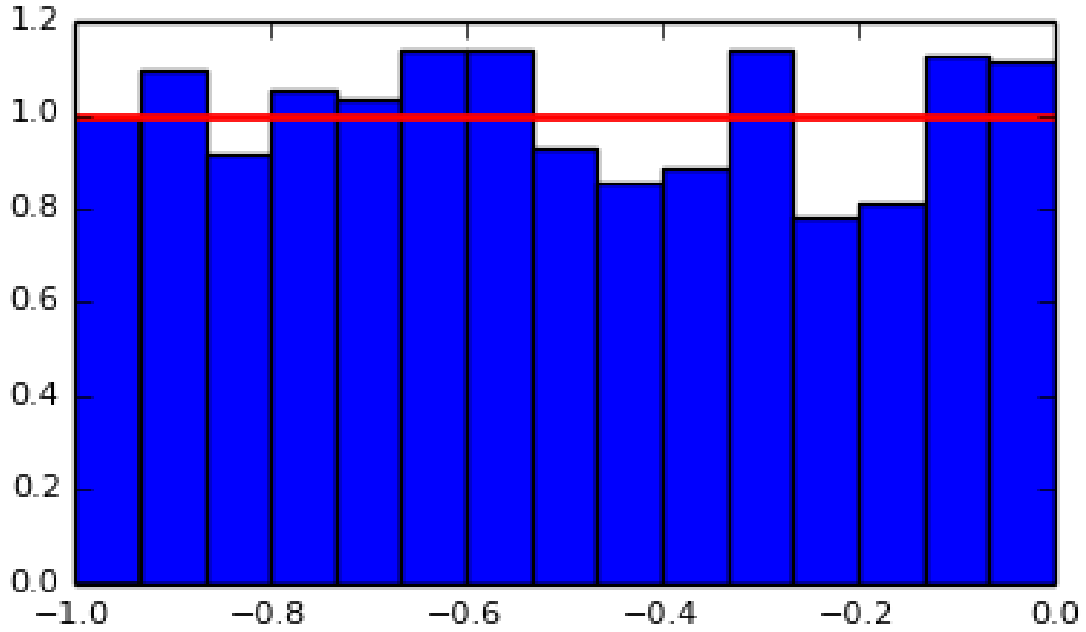
$$S_I \sim N^{-1/2}$$

Distributions

Python: Random number generators

```
import numpy as np
import matplotlib.pyplot plt
for i in range(1000):
   xn= np.random.uniform()
   plt.hist(xn)
plt.plot(bins, np.ones_like(bins),linewidth=2,
  color='r')
plt.show()
```

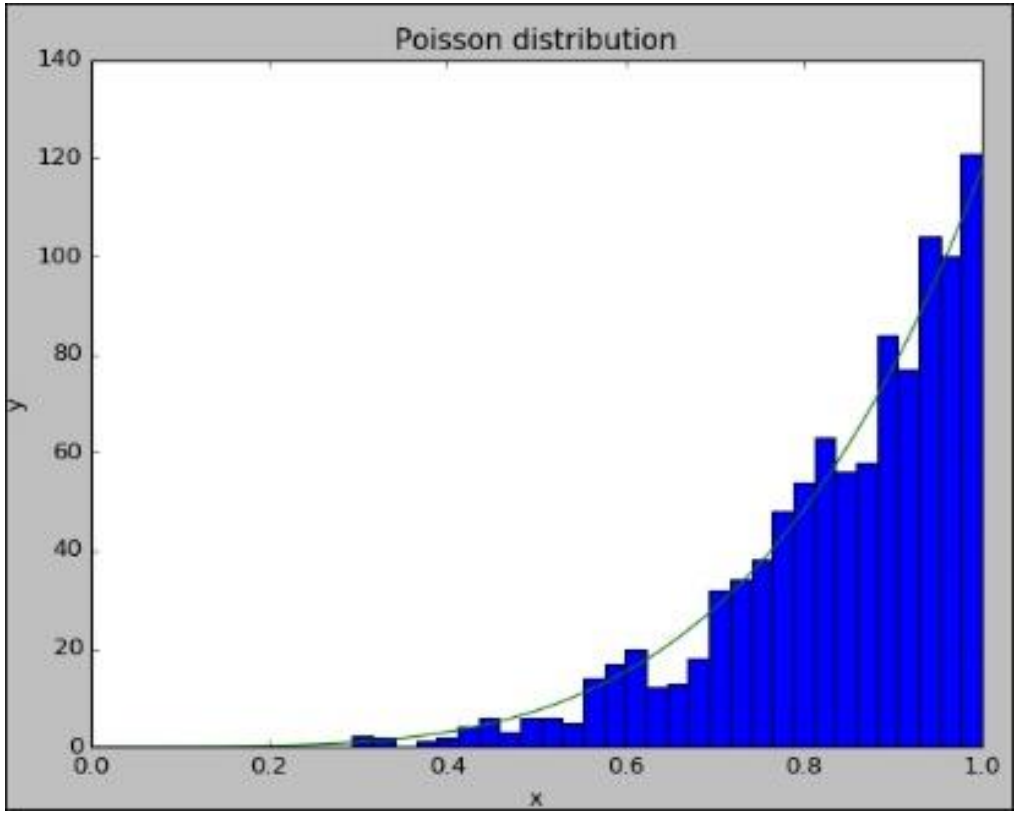
Python: Random number generators



Generating random numbers from a Poisson distribution

```
import numpy as np
import scipy as sp
import matplotlib.pyplot as plt
x=sp.random.poisson(lam=1, size=100)
#plt.plot(x,'o')
a = 5. # shape
n = 1000
s = np.random.power(a, n)
count, bins, ignored = plt.hist(s, bins=30)
x = np.linspace(0, 1, 100)
y = a*x**(a-1.)
normed_y = n*np.diff(bins)[0]*y
plt.title("Poisson distribution")
plt.ylabel("y")
plt.xlabel("x")
plt.plot(x, normed_y)
plt.show()
```

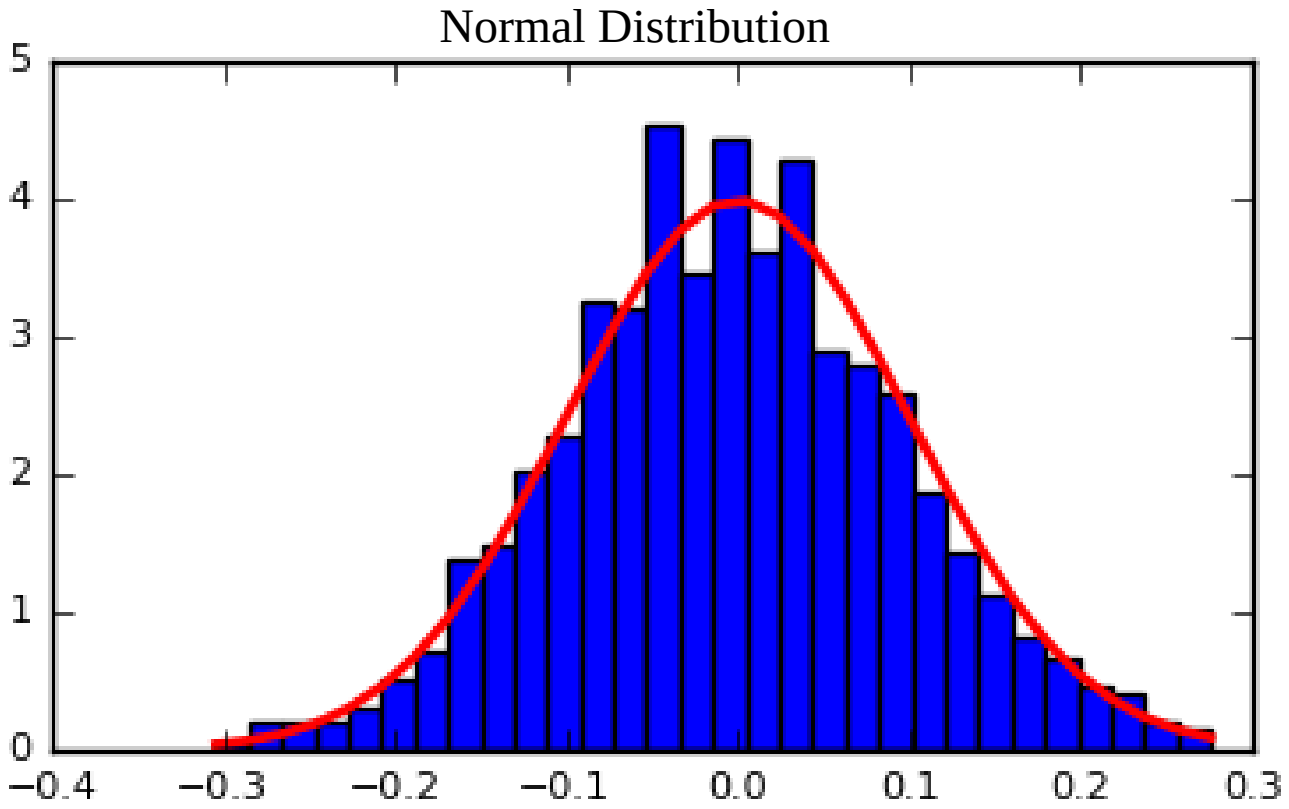
Generating random numbers from a Poisson distribution



Generating random numbers from a Normal distribution

```
import matplotlib.pyplot as plt
import numpy as np

mu, sigma = 0.5, 0.1
s = np.random.normal(mu, sigma, 1000)

# Create the bins and histogram
count, bins, ignored = plt.hist(s, 30, normed=True)

plt.plot(bins,1/(sigma*np.sqrt(2 * np.pi))*np.exp(-(bins - mu)**2/(2*sigma**2))
,linewidth=2, color='r')

plt.show()
```

Generating random numbers from a Normal distribution



Monte Carlo Generator

MC programs contain cross sections of many different processes, that are generated (one or many) + eventually some cuts on phase space applied.

the generator produces events with particles from a certain process, distributed according to the relevant cross section

For complicated physics, such as hadron production, the whole process is subdivided into sub-steps, simulated according to more or less well known models

Most popular ones: PYTHIA, HERWIG

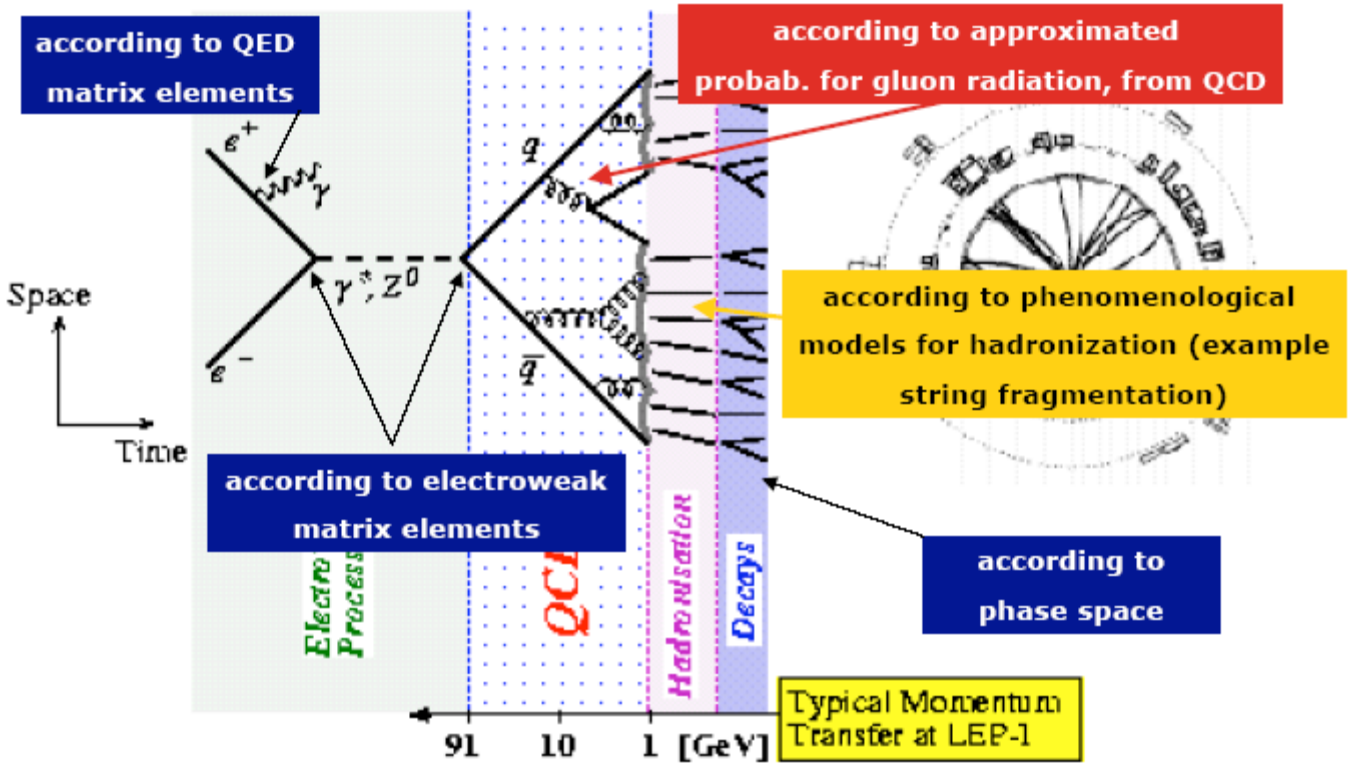
- contain an enormous list of processes (QED, QCD, new physics,...)

Very precise generators for electro-weak fermion production: KORALZ, KORALW

Generator for Higgs production in e^+e^- annihilation: HZHA

Generators for Deep Inelastic scattering : DJANGO, CASCADE, RAPGAP, ...

Monte Carlo Generator



Monte Carlo event generators

1. $e^+e^- \rightarrow$
2. El.Weak$\rightarrow Z^0 \rightarrow s\ s^{\sim}$
3. QCD $s \rightarrow s\ g$

   $s^{\sim} \rightarrow s^{\sim} g$

   $g \rightarrow u\ u^{\sim}$

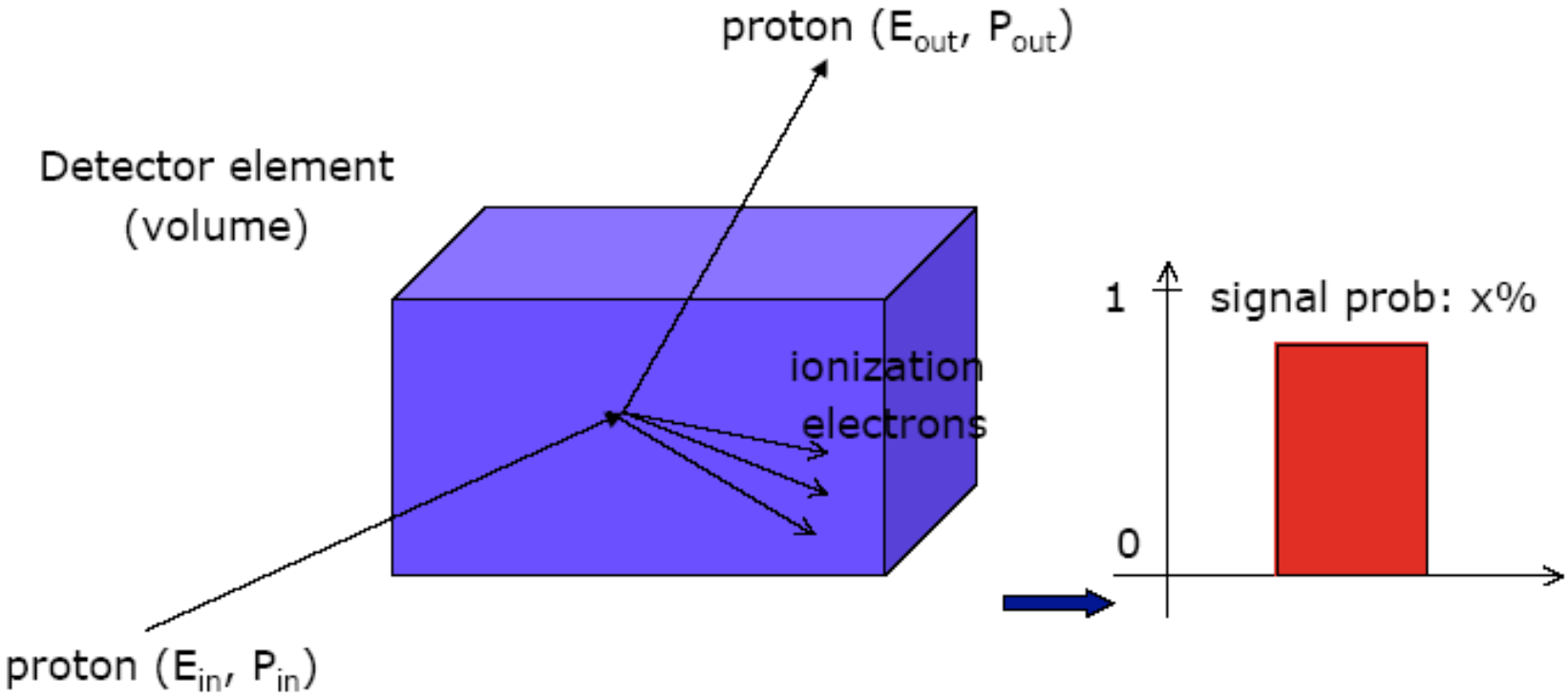
4. Hadronisation

Example : Pythia event listing

I	particle/jet	KS	KF	orig	p_x	p_y	p_z	E	m
1	!e-!	21	11	0	0.000	0.000	45.600	45.600	0.001
2	!e+!	21	-11	0	0.000	0.000	-45.600	45.600	0.001
3	!e-!	21	11	1	0.000	0.000	45.600	45.600	0.001
4	!e+!	21	-11	2	0.000	0.000	-45.600	45.600	0.001
5	!e-!	21	11	3	0.000	0.000	45.600	45.600	0.000
6	!e+!	21	-11	4	0.000	0.000	-45.600	45.600	0.000
7	!Z0!	21	23	0	0.000	0.000	0.000	91.200	91.200
8	!s!	21	3	7	-39.289	20.658	10.437	45.600	0.199
9	!s~!	21	-3	7	39.289	-20.658	-10.437	45.600	0.199
10	(Z0)		11	23	7	0.000	0.000	0.000	91.200	91.200
11	(s)	A	12	3	8	-25.977	16.144	8.944	31.867	0.199
12	(g)	I	12	21	8	-5.588	0.453	-0.559	5.634	0.000
13	(u~)	V	11	-2	9	4.109	-16.499	5.349	17.824	0.006
14	(s~)	A	12	-3	9	24.487	3.688	-16.595	29.818	0.199
15	(g)	I	12	21	9	1.245	-0.237	0.148	1.276	0.000
16	(u)	V	11	2	9	1.724	-3.549	2.713	4.789	0.006
17	(string)		11	92	11	-27.456	0.098	13.734	55.325	46.026
18	(K*~0)		11	-313	17	-9.946	5.625	3.716	12.851	0.922
19	(f_2)		11	225	17	-7.974	5.444	2.544	10.053	1.165
20	(rho-)		11	-213	17	-4.402	1.720	0.385	4.828	0.896
21	(rho+)		11	213	17	-2.498	1.398	0.920	3.878	0.690
22	pi-		1	-211	17	-5.408	2.263	0.899	5.933	0.140
23	K+		1	321	17	-0.187	-0.241	0.085	0.587	0.494
24	(K*-)		11	-323	17	-0.518	-0.243	-0.057	1.848	0.866
25	(f_1)		11	20223	17	-0.351	-1.380	0.394	1.934	1.248
26	(f_1)		11	20223	17	3.820	-14.488	4.936	15.829	1.301
27	(string)		11	92	14	27.456	-0.098	-13.734	36.675	18.562
28	K+		1	321	27	16.291	2.297	-10.932	19.759	0.494
29	(h_1)		11	10223	27	5.228	1.174	-2.899	6.228	1.292
30	K-		1	-321	27	2.582	-0.543	-1.868	3.208	0.494
31	(K0)		11	311	27	8.731	-0.191	0.416	8.996	0.498
32	(K*~0)		11	-313	27	0.484	0.178	-0.432	1.121	0.897
33	(K0)		11	311	27	1.824	-0.521	0.668	1.919	0.498
34	(rho+)		11	213	27	1.196	-2.491	1.312	3.145	0.729
35	(K-0)		11	-311	18	-5.453	3.425	2.041	6.773	0.498

etc etc

Monte Carlo detector simulation



⇒ subdivide detector in many sub-elements, and track every particle step by step through volumes

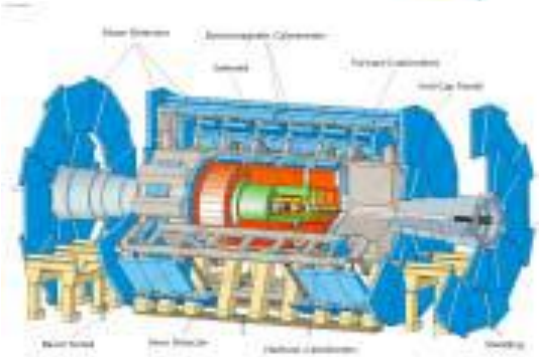
Monte Carlo detector simulation

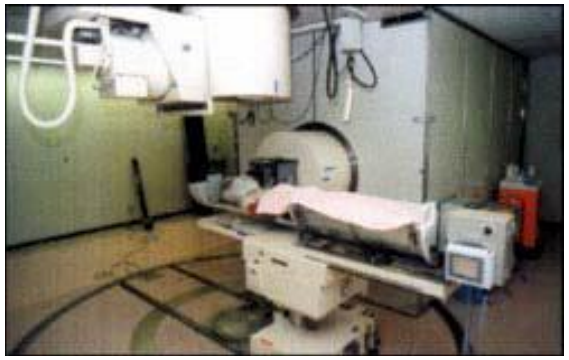
- The reaction of a particle with the detector material is according some physics process
- So it can again be simulated using a Monte Carlo generator
- Have to
  - Set up geometry of the detector
  - Define material
  - Define the possible types of interactions e.g. Bremsstrahlung, ionization etc..
  - Simulate the signal generation in an active detector element

What is Geant4?

- **A Monte Carlo software toolkit to simulate the passage of particles through matter.**

- It is for detector simulation of research in
  - **High energy physics**
  - **Nuclear physics**
  - **Cosmic ray physics**
  - **Medical Physics**
- It is also for application in
  - **Space science**
  - **Radiological science**
  - **Radiation background calculation**
  - **etc**
- ATLAS/LHC
- CMS/LHC
- HIMAC/NIRS INTEGRAL/ESA

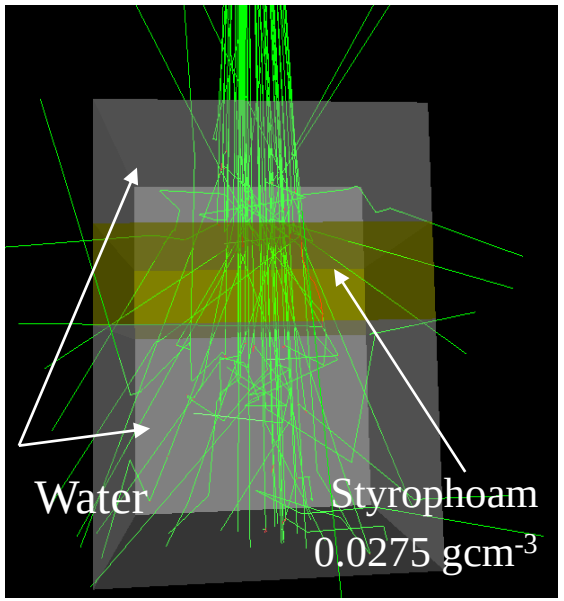


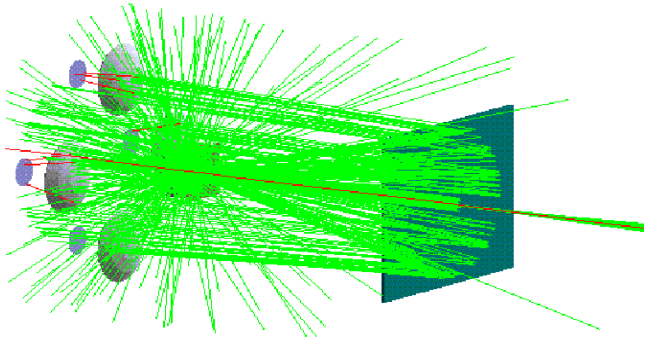


installation/tutorials :
http://geant4.web.cern.ch/

Geant4

- **General characteristics of a particle detector**
  - **simulation program:**
    - You specify the geometry of a detector.
    - Then the program automatically *transports* the particle you injected to the detector by simulating the particle interactions in matter based on the Monte Carlo method
  - **The heart of the simulation**
    - The Monte Carlo method to simulate the particle interactions in matter





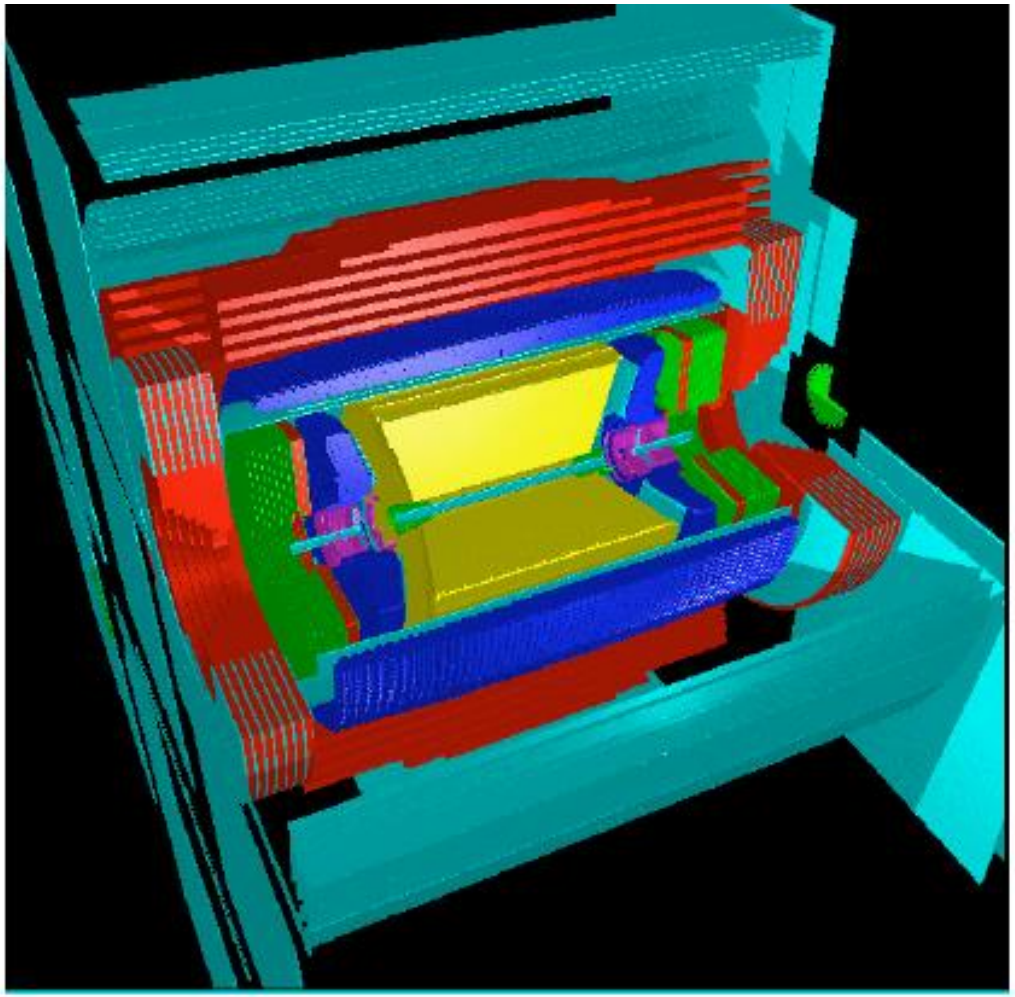
V. Good tutorial

https://www.ge.infn.it/geant4/

Geant4

OPAL detector :

GEANT layout of material



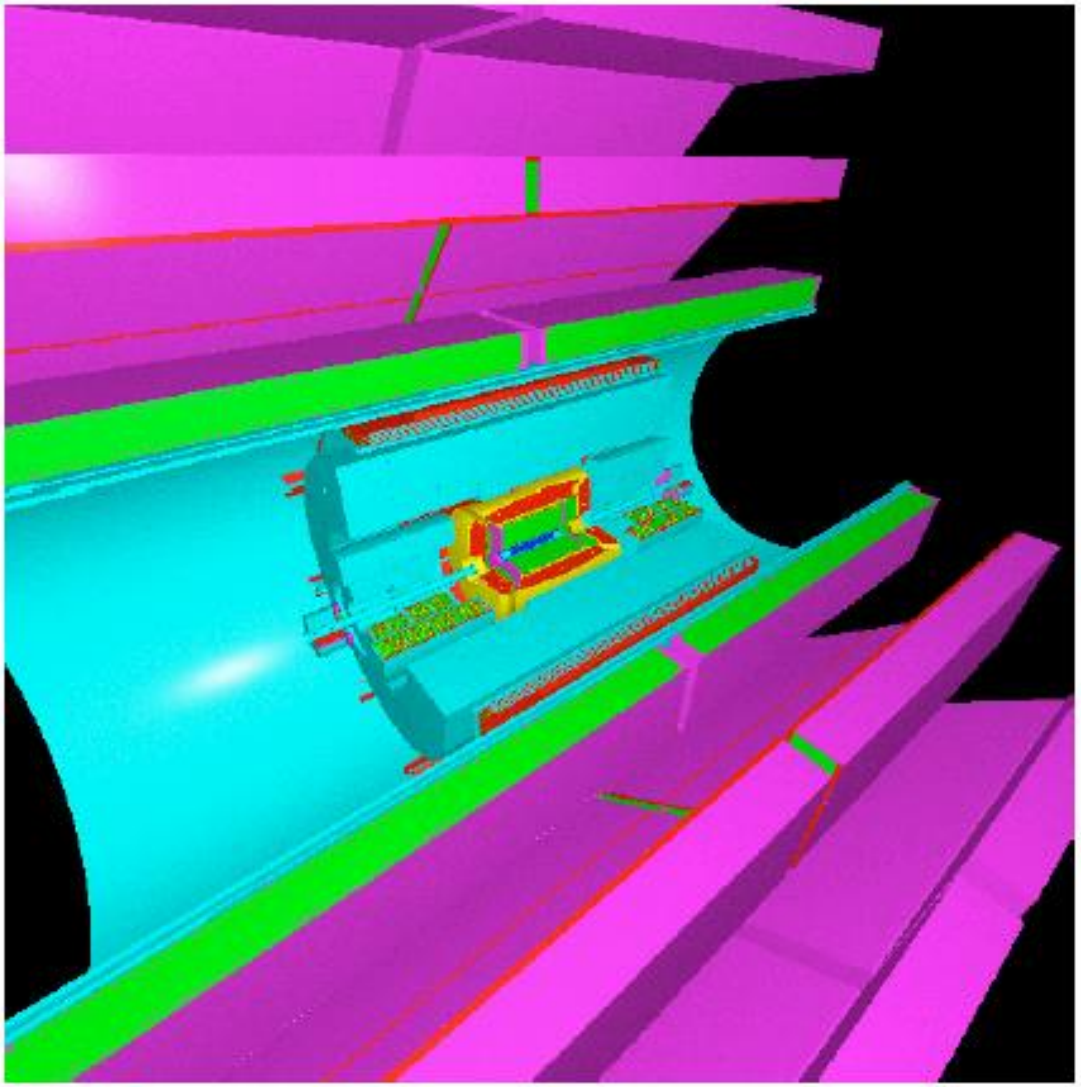
Geant4

L3 detector :

GEANT layout of material

Note:

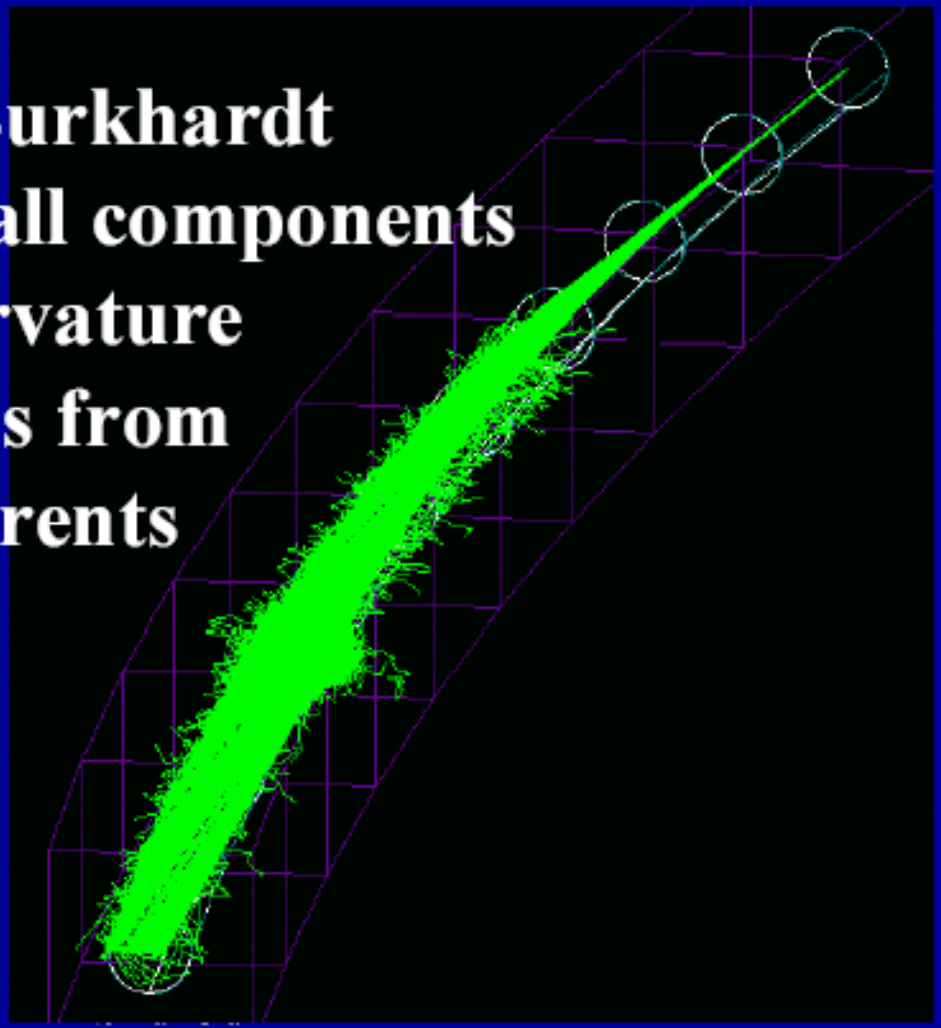
material like muon chambers are just "cylinders" with certain properties.



Geant4

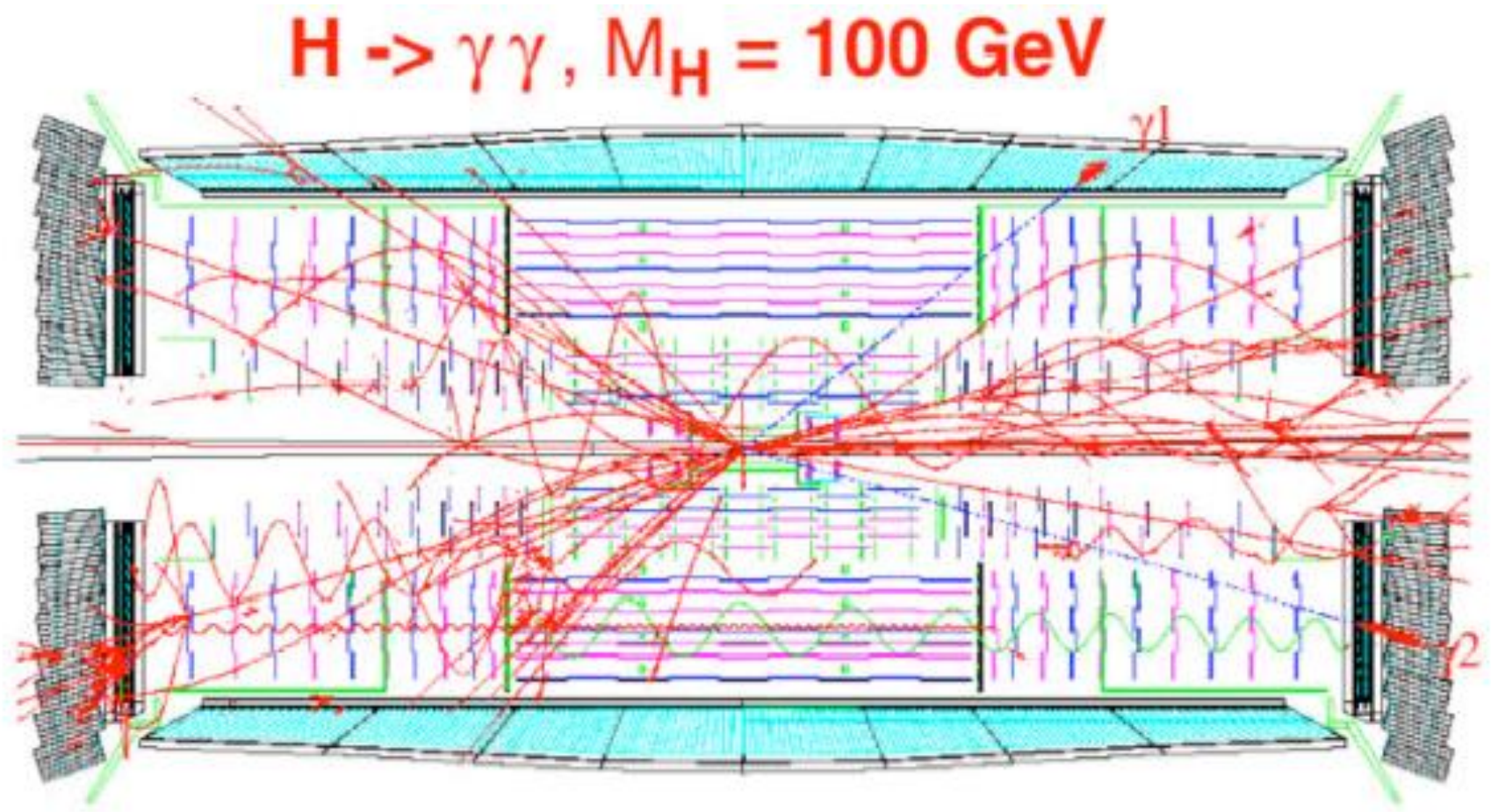
Synchrotron Radiation

Generator of H. Burkhardt
Implemented for all components
Based on local curvature
Individual photons from individual parents



2/22/2002

Geant4



standard program package : GEANT

In Class Exercise

- Write a python script to generate MC events distributed according to PDFs
- Produce 10,000 events in range of [0,1] for pdf A and pdf B
- Plot both distribution on the same canvas

$$P_A(x) = \frac{2}{1-e^{-2}} e^{-2x} \qquad P_B(x) = 3x^2$$