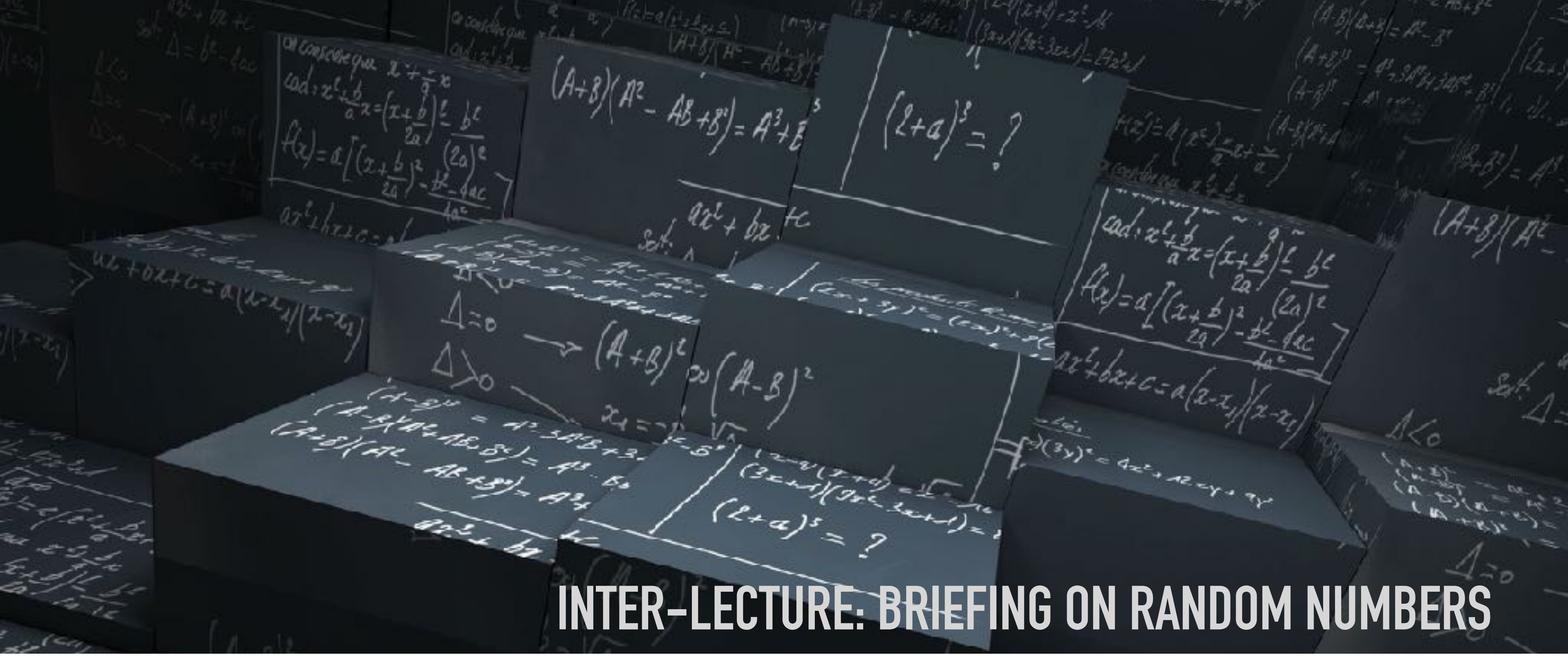


INTER-LECTURE: BRIEFING ON RANDOM NUMBERS

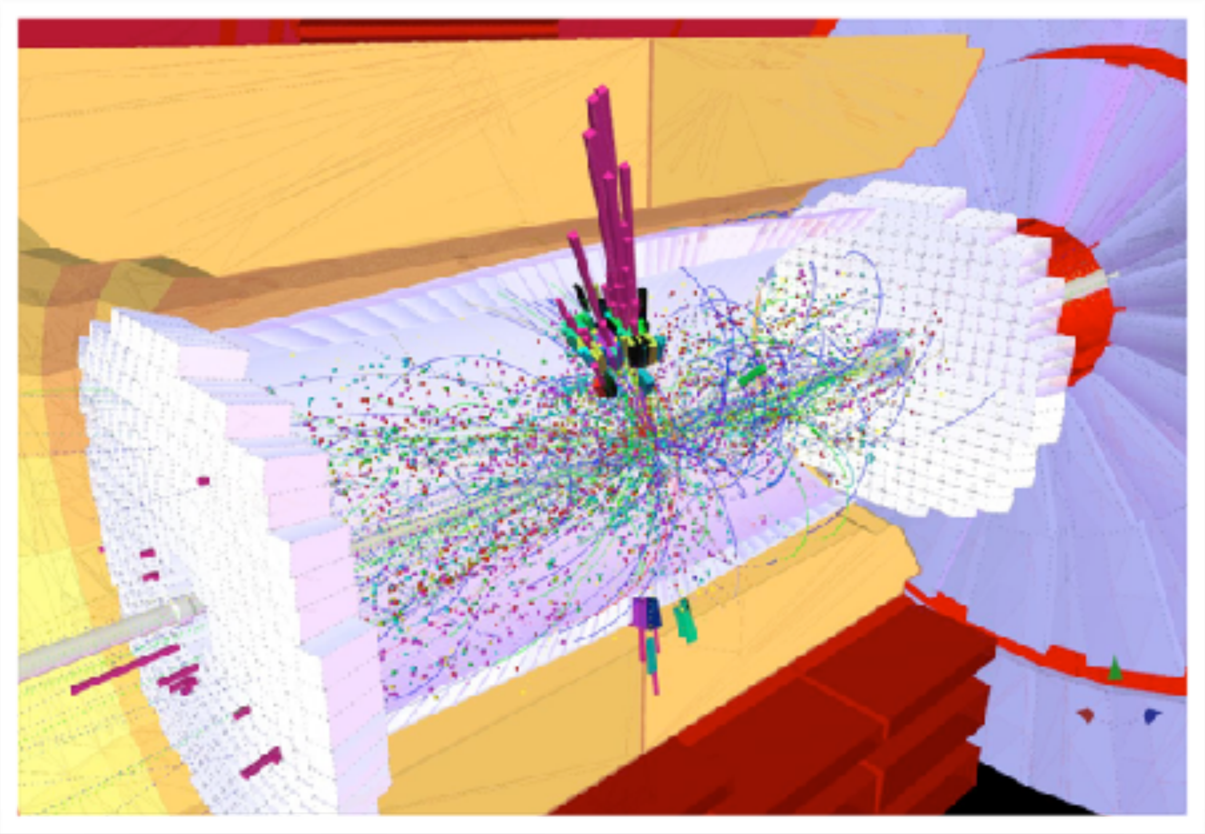
STATISTICAL ANALYSIS IN EXPERIMENTAL PARTICLE PHYSICS

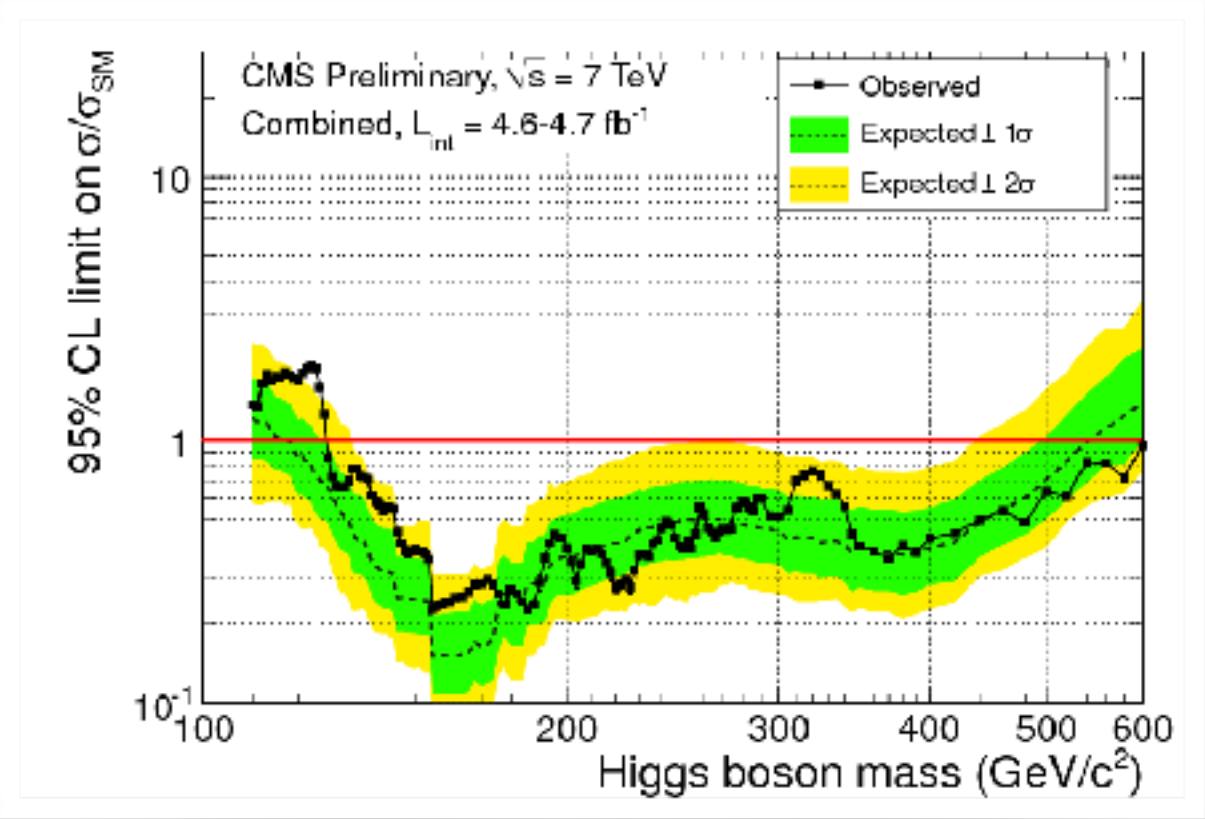
Kai-Feng Chen

National Taiwan University

RANDOM NUMBERS

- The computing generated random numbers provide a way to study the statistical properties for any model you would propose?
- In high-energy physics: heavily used in Monte Carlo simulations, or statistical tests!
- Understand the limit of your random number generator is very important.




CMS Preliminary, √s = 7 TeV
Combined, L_{int} = 4.6-4.7 fb^{-1}
Observed
Expected ± 1σ
Expected ± 2σ
95% CL limit on σ/σ_{SM}
10
1
10^{-1}
100
200
300
400
500
600
Higgs boson mass (GeV/c^2)

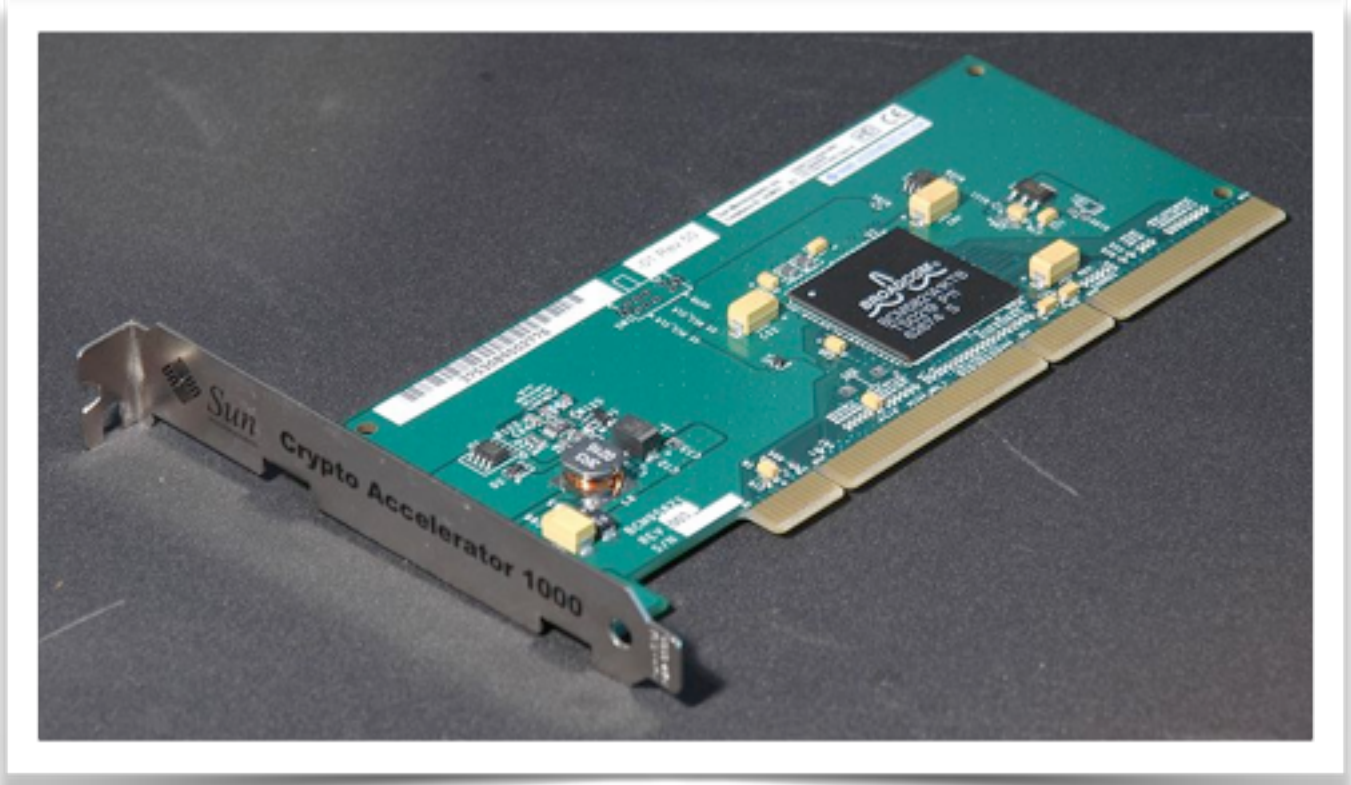
"TRUE" VERSUS "PSEUDO" RANDOM NUMBER GENERATORS

- *Quote from Wikipedia:*
  "There are two principal methods used to generate random numbers. One measures some physical phenomenon that is expected to be random and then compensates for possible biases in the measurement process. The other uses computational algorithms that can produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a **seed** or **key**. The latter type are often called **pseudo-random number generators**."

Carefully chosen pseudo-random number generators can be used in many applications instead of true random numbers!

HARDWARE RANDOM NUMBERS GENERATORS

- A hardware random number generator can generates random numbers from a physical process, rather than a computer program, e.g. such devices can generate statistically random “noise” signals, such as **thermal noise**, the **photoelectric effect**, etc.



Such a device is usually very useful for cryptographic work; for most of our scientific works (no security issue!) the pseudo random number generators are already good enough.

BASIC PROPERTIES OF (PSEUDO) RANDOM NUMBERS

- Running a random number generator should result a sequence of "good" random numbers: $X_1, X_2, \ldots, X_N, \ldots$
- The elements should be independent and identically distributed, i.e.:
  - $P(X_i) = P(X_j)$
  - $P(X_n | X_{n-1}) = P(X_n)$
- Pseudo random number generators are nothing more than a recursive algorithm (which is relatively simple and quick), that can "process" your input seed to a series of numbers.
- A pseudo random number generator always has a **limited "period"**. The "randomness" of the random number generator cannot be guaranteed after that!

RANDOM NUMBER GENERATORS IN ROOT

- The basis generator interface: TRandom. It should not be used directly but used via its derived classes (TRandom1/TRandom2/TRandom3):
  - **TRandom**: very fast but a BAD generator, period: 10^9
  - **TRandom1**: “RANLUX” slow but long period: 10^{171}
  - **TRandom2**: “Tausworthe” very fast, okay period: 10^{26}
  - **TRandom3**: “Mersenne Twister” fast, very long period: 10^{6000}
- All these classes have the same functionalities for generating uniform distributions (in integer, float point numbers) or many commonly used distributions (exponential, Gaussian, etc.)
- Generally **TRandom3** is the recommended one (obvious reason!).
- Also GSL based generators available with ROOT::Math.

MERSENNE TWISTER

- It was developed in 1997 by Makoto Matsumoto (松本真) and Takuji Nishimura (西村拓士).
- Webpage (*you can also download the source code*): http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
- It is claimed to be fast, with a period of $2^{19937}-1$ (~10^{6001}).
- This may not be the generator with the longest period in the world, but it's a very famous one.
- There are several variants released by the authors recently: SFMT (*SIMD-oriented Fast MT*), MTGP (*MT on graphic processor*), TinyMT (*light weighted version*), etc.
- Default algorithm in GSL library as well.

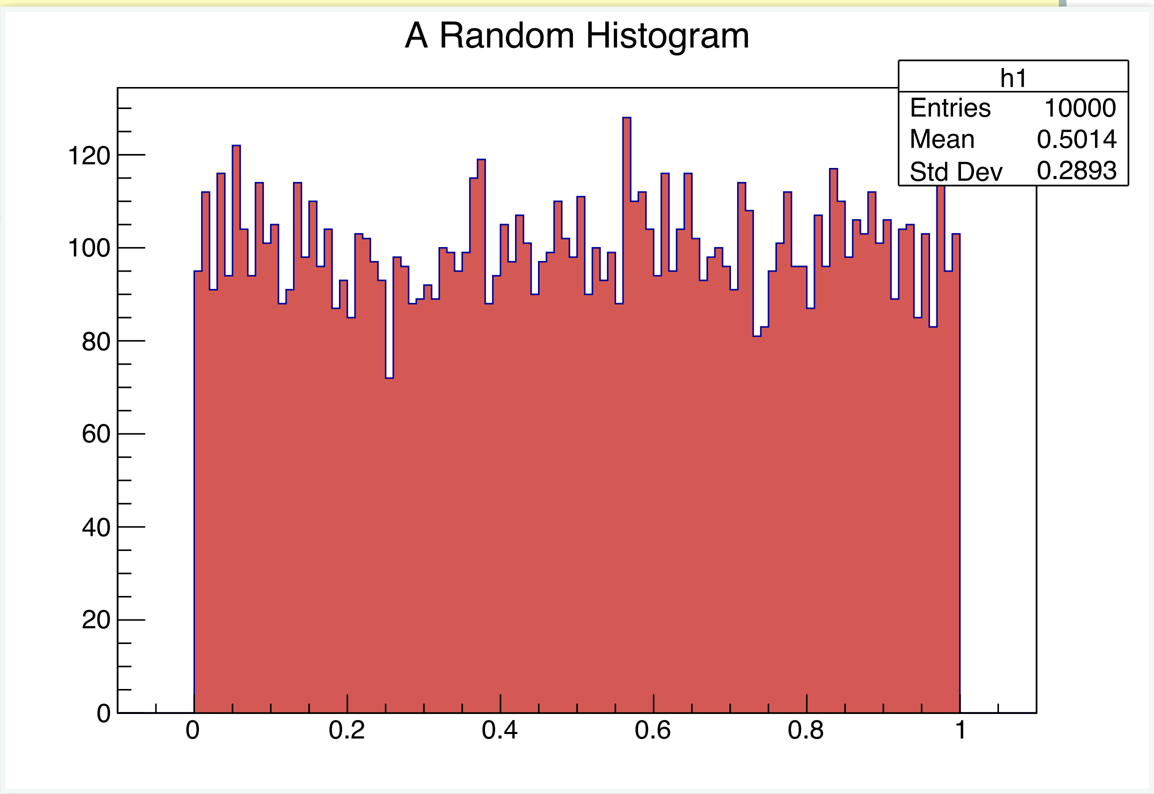
USING TRANDOM CLASSES

- The most straightforward call to TRandomN classes can be demonstrated as following:

example_01.cc

```
{
    TRandom3 rnd(1234);

    TH1D *h1 = new TH1D("h1","A Random Histogram",120,-0.1,1.1);

    for(int evt=0; evt<10000; evt++) h1->Fill(rnd.Rndm());

    h1->SetFillColor(50);
    h1->Draw();
}
```

Seed

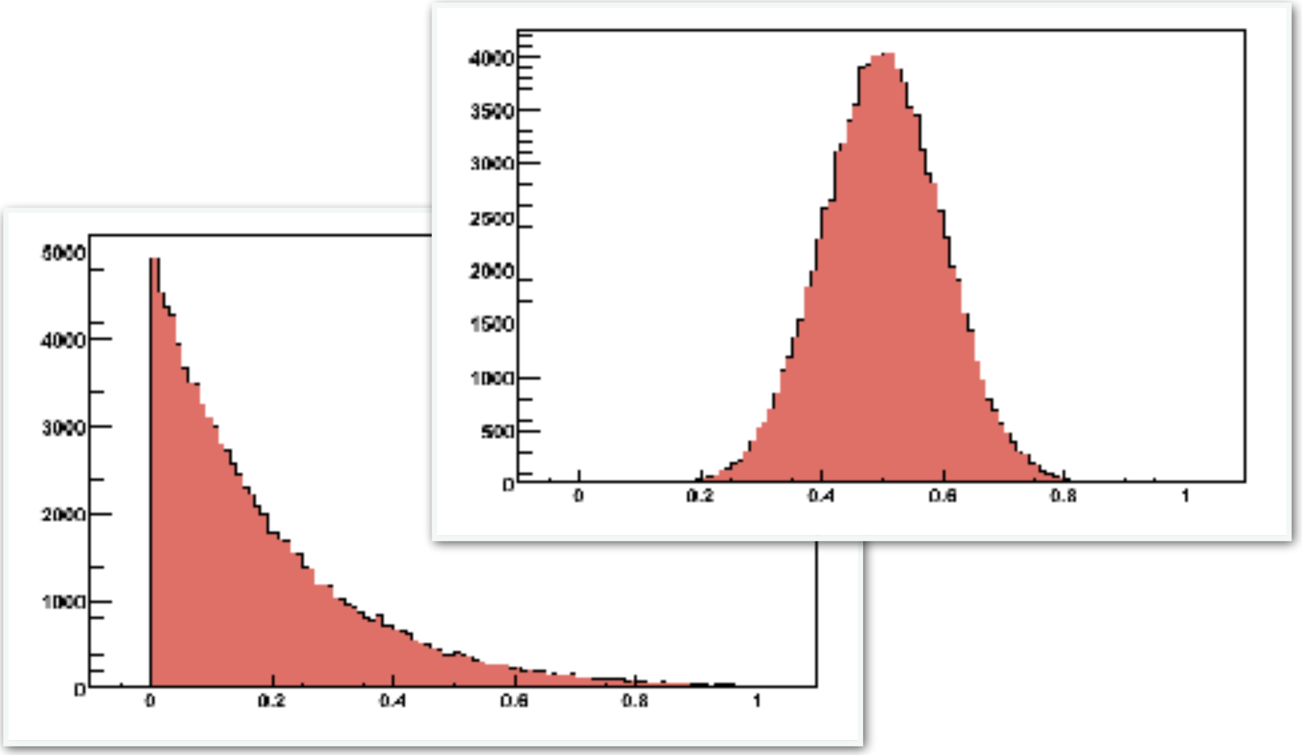


- **Rndm()** is the simplest call to produce uniformly-distributed floating points between 0 and 1.

- **Uniform()** is also available.

OTHER DISTRIBUTIONS

- All the TRandomN classes have the following implementations:
  - **Exp(tau)**
  - **Integer(imax)**
  - **Gaus(mean,sigma)**
  - **Rndm()**
  - **Uniform(x1)**
  - **Landau(mpv,sigma)**
  - **Poisson(mean)**
  - **Binomial(ntot,prob)**



It is very straightforward to change the distribution in the previous example code! You are encouraged to try them!

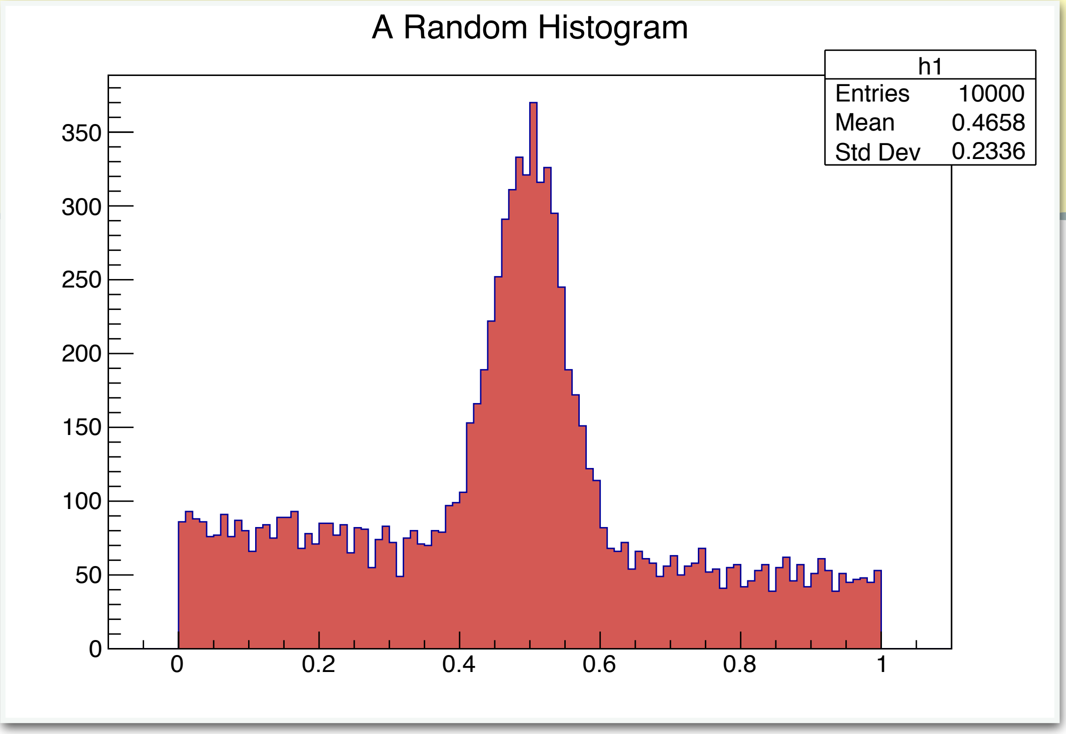
- But how about the distributions not (*yet*) implemented?

A QUICK SOLUTION WITHIN ROOT

- If you want to generate the random numbers according to a user-defined function, and with fixed lower/upper bounds, it is very convenient to use TF1.

example_02.cc

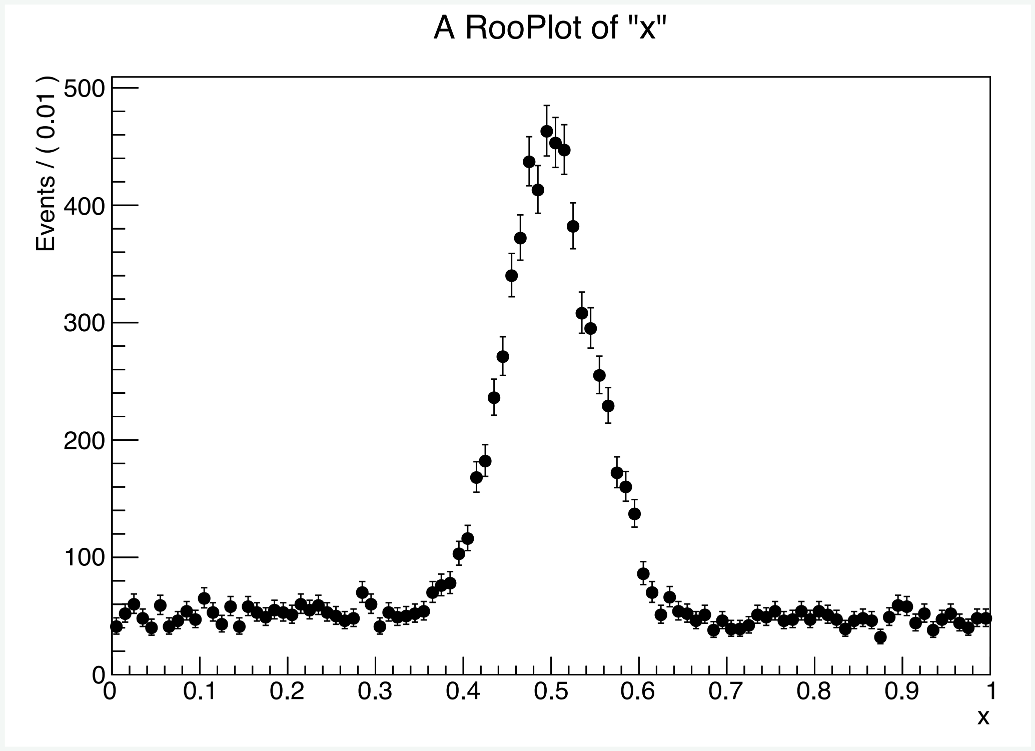
```
{
    TH1F *h1 = new TH1F("h1","A Random Histogram",120,-0.1,1.1);

    TF1 *f1 = new TF1("f1","[0]+[1]*x+[2]*gaus(2)");
    f1->SetParameters(2.,-1.0,2.5,0.5,0.05);

    for(int evt=0; evt<10000; evt++) h1->Fill(f1->GetRandom());

    h1->SetFillColor(50);
    h1->Draw();
}
```



You already see this example in the previous lecture in fact!

WITH THE ROOFIT

- RooFit also provides the tool to generate any distribution that can be implemented within RooFit framework.
- We will discuss RooFit more in the upcoming lectures!



example_03.cc

```
{
    using namespace RooFit;

    RooRealVar x("x","x",0.,1.);

    RooGaussian Gauss("Gauss",
            "Gaussian PDF",x,RooConst(0.5),RooConst(0.05));
    RooExponential Exp("Exp","exponential PDF",x,RooConst(-0.1));
    RooAddPdf model("model","joint model",Gauss,Exp,RooConst(0.5));

    RooDataSet *data = model.generate(x,10000);
    RooPlot *frame = x.frame();

    data->plotOn(frame);
    frame->Draw();
}
```

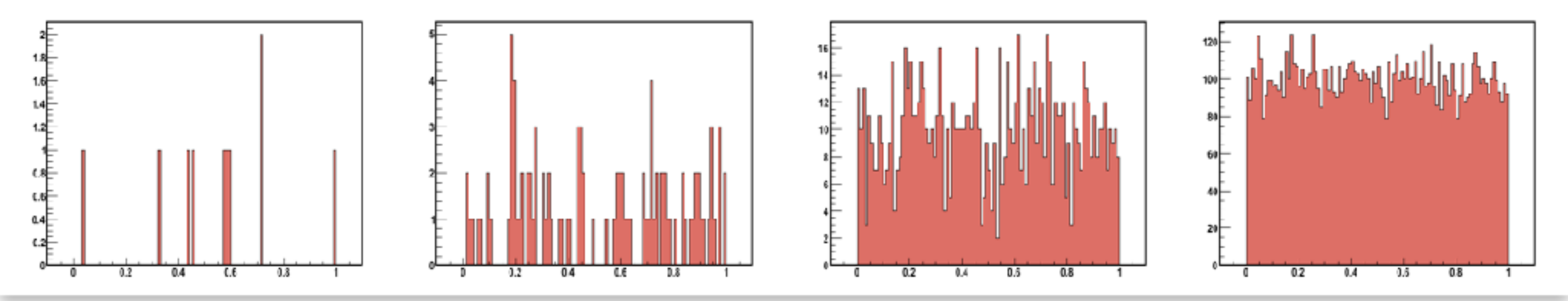
OR, DO THINGS BY OURSELVES

- You may think since the tools are already very convenient, why we still need to learn how to **generate (user) random distributions by ourselves**?
- In fact this helps (*a lot*) to understand how these convenient tool works and the limitation of them!
- It is a pretty common case, if one finds the existing tool does not produce the correct results, you will understand "why" immediately.

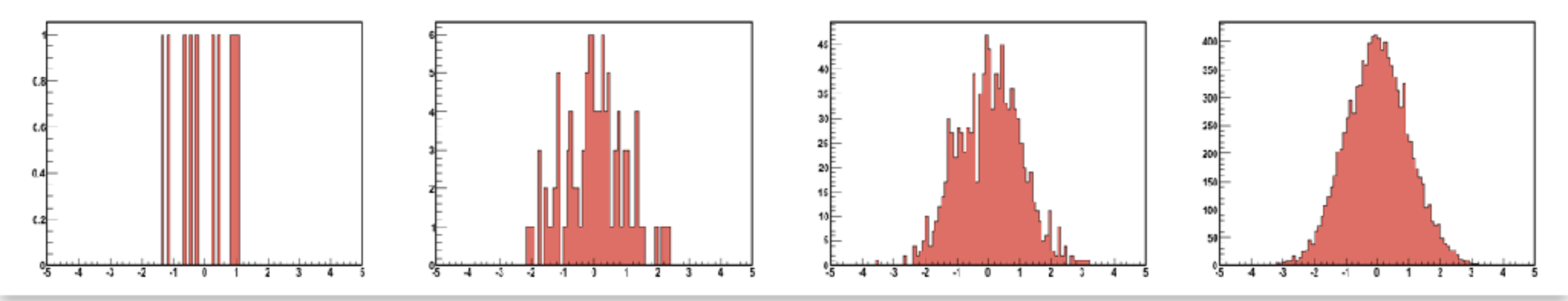
Yes, it is very important to understand what you are actually doing!

ACCUMULATE RANDOM DISTRIBUTIONS

- ➤ Assuming we can generate uniformly random distribution already, for example, using **TRandom3::Rndm()**:



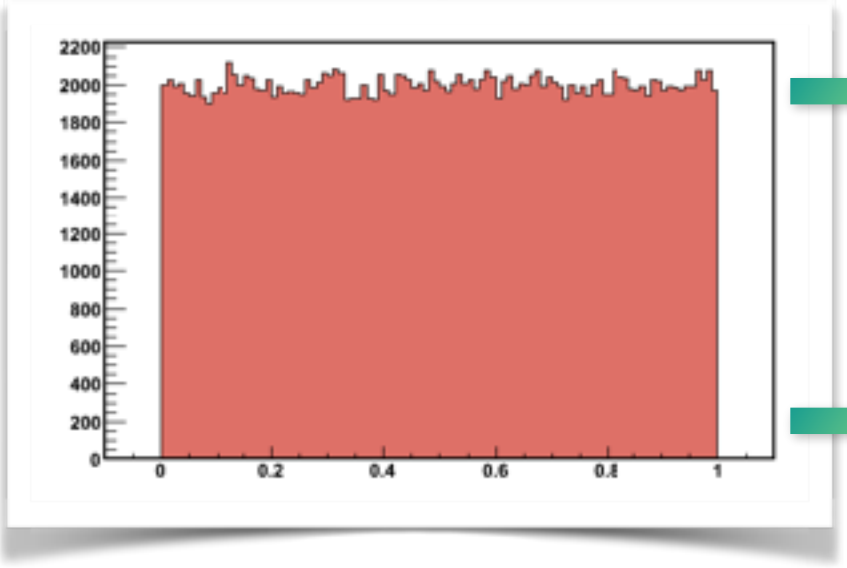
- ➤ How to generate any non-uniform distribution, such as a **Gaussian**?

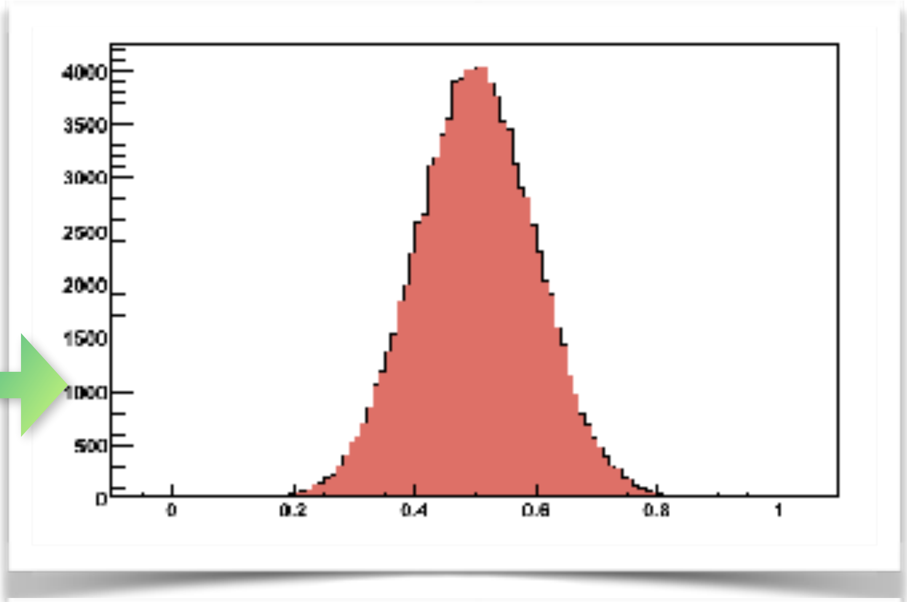


GENERATION OF NON-UNIFORM DISTRIBUTIONS

➤ Probably this is the case that we are usually facing: covert uniform random distributions with your own code.

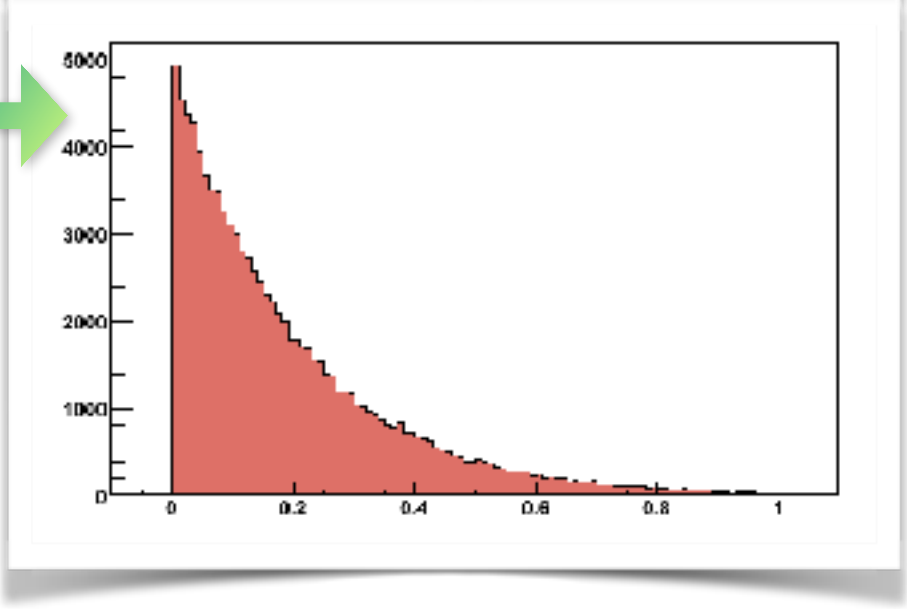
Uniform





Gaussian

$$f(x) = \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$



Exponential

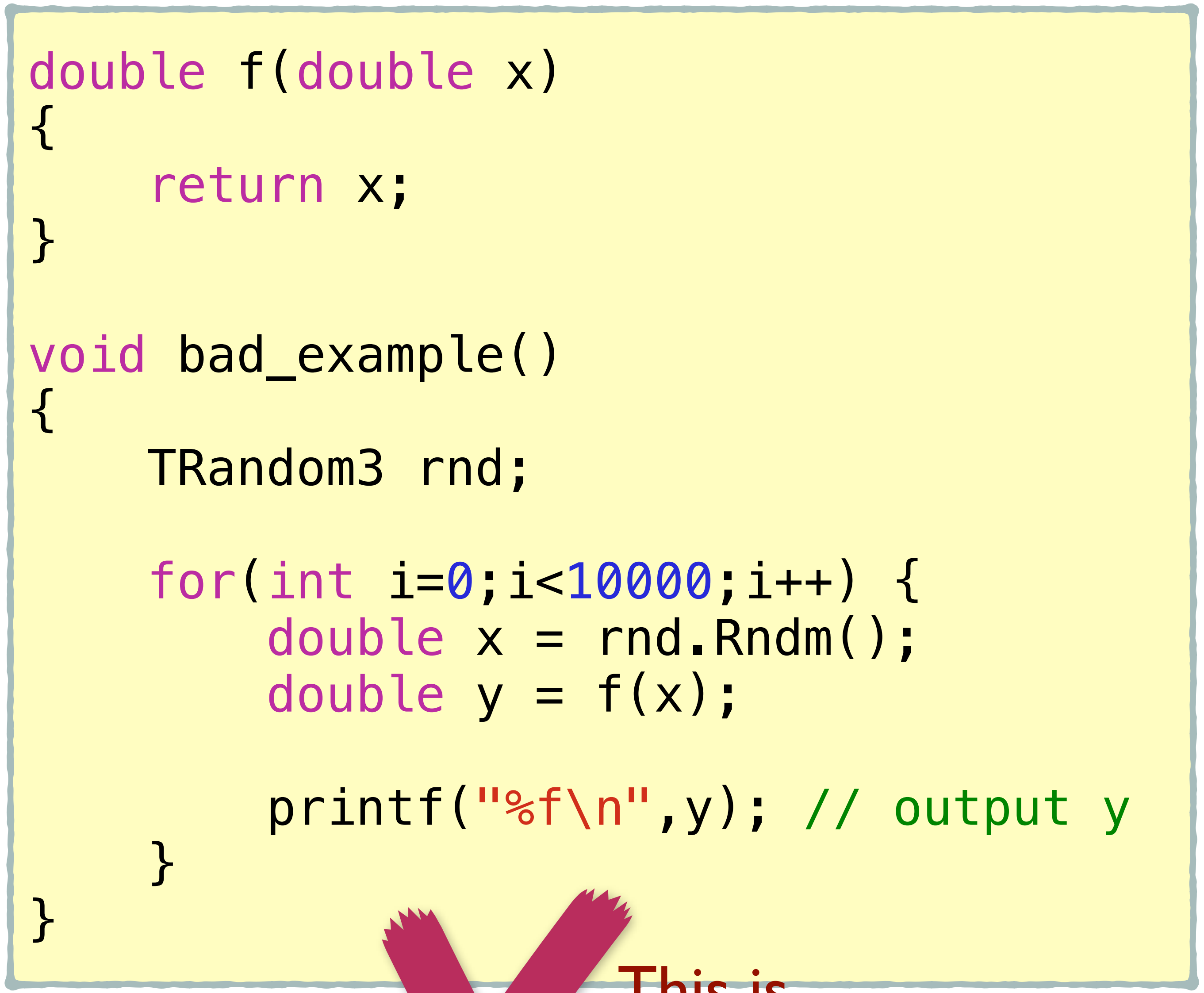
$$f(x) = \exp\left(-\frac{x}{\tau}\right)$$

GENERATION OF NON-UNIFORM DISTRIBUTIONS (II)

- Suppose you already have a defined function **f(x)**, all positive in the range of **[0,1]**. For example a simple function like:

$$f(x) = x$$

- Now we want to generate a random distribution based on this function f(x). Probably 90% of the people will start with such a naive code?

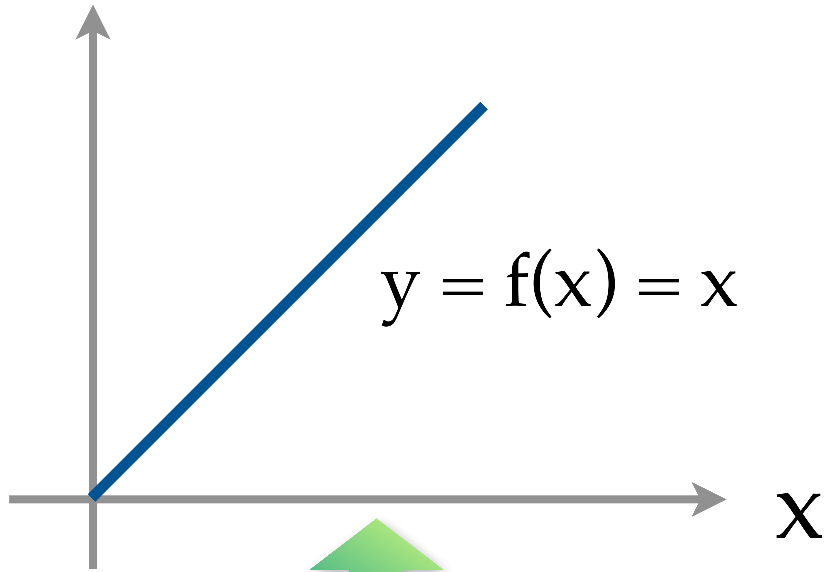


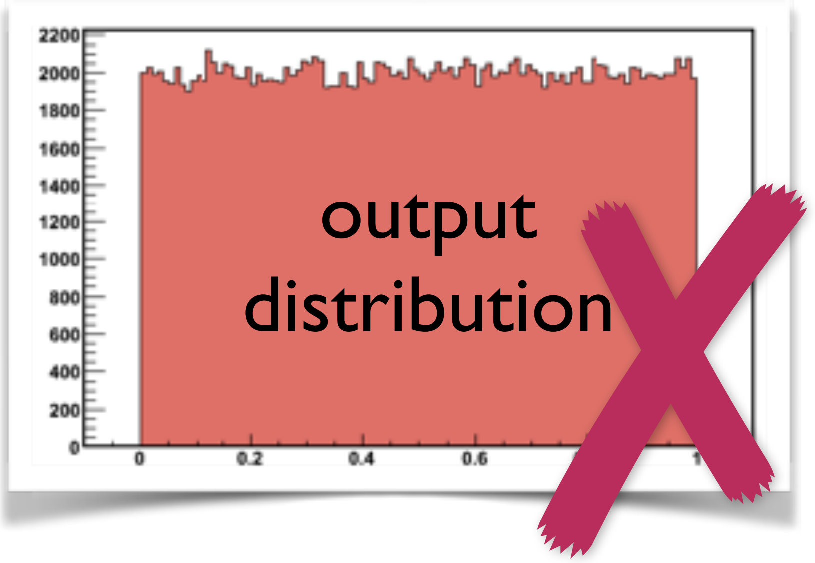
```
double f(double x)
{
    return x;
}

void bad_example()
{
    TRandom3 rnd;

    for(int i=0;i<10000;i++) {
        double x = rnd.Rndm();
        double y = f(x);

        printf("%f\n",y); // output y
    }
}
```

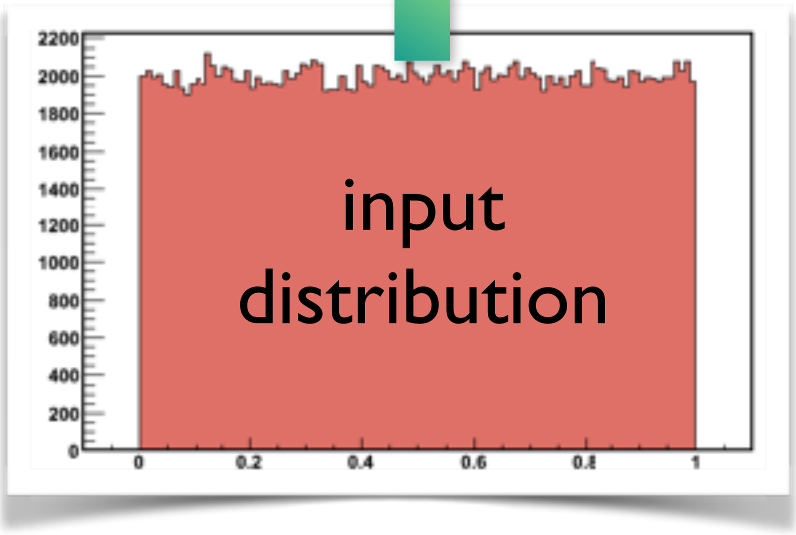
This is definitely incorrect...

GENERATION OF NON-UNIFORM DISTRIBUTIONS (III)

- Actually the given function, *y=f(x)*, only gives the **weights** as a function of x; it does not produce a random distribution by simply inserting a random distribution along x.







Since y = f(x) = x, the output distribution is just a replica of the input distribution x.

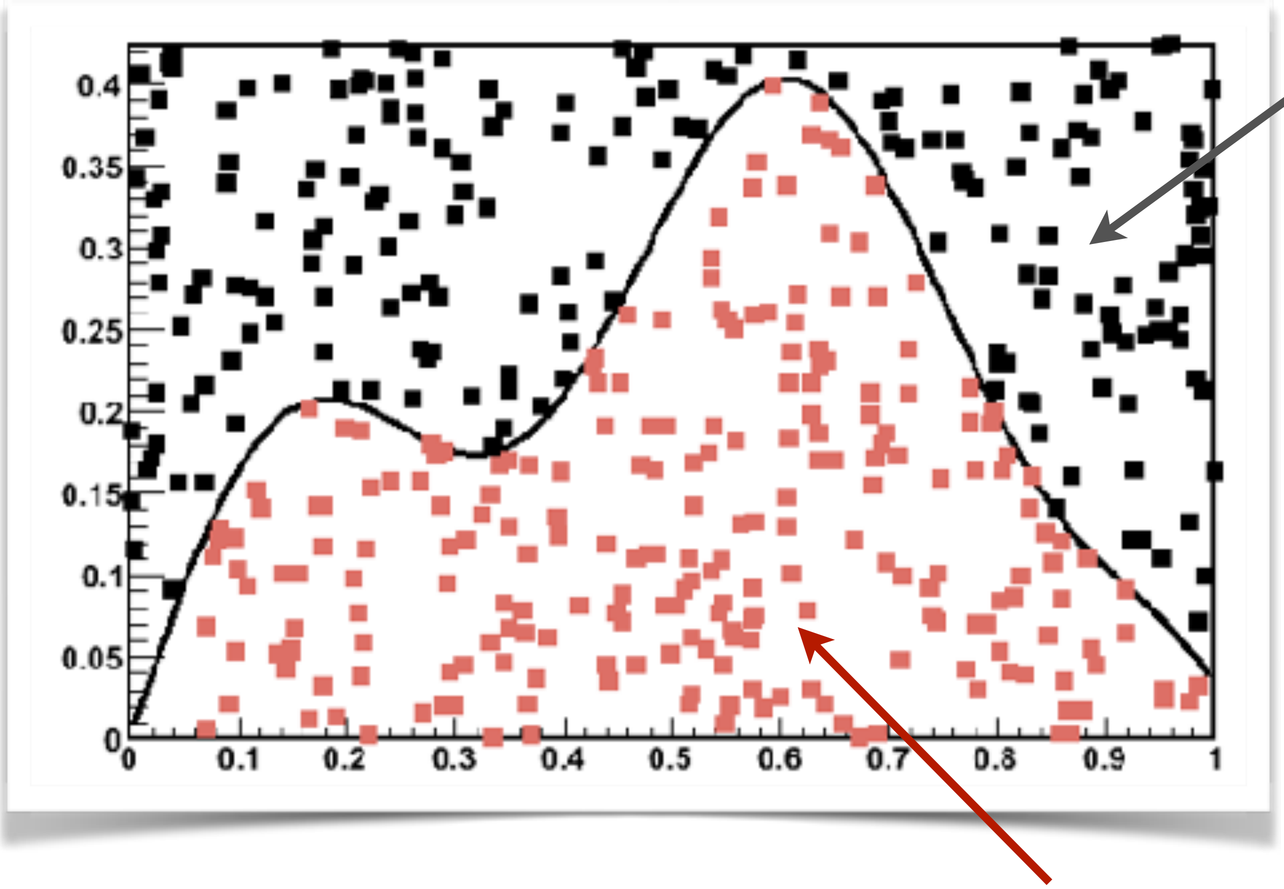
START FROM THE SIMPLEST WAY

- The most simple **"Hit-or-Miss"** method (Von Neumann rejection) is actually quite similar to spay cinnamon powder on your cappuccino...

THE HIT-OR-MISS METHOD

- This is one of the most simplest algorithms, it's quite inefficient, but still very useful! The trick is simply: **instead of 1D –– generate the random numbers uniformly in 2D**:

```
Generate
$x$ = rand()
$y$ = rand()
if $y < f(x)$: accept $x$
```



REJECT

ACCEPT

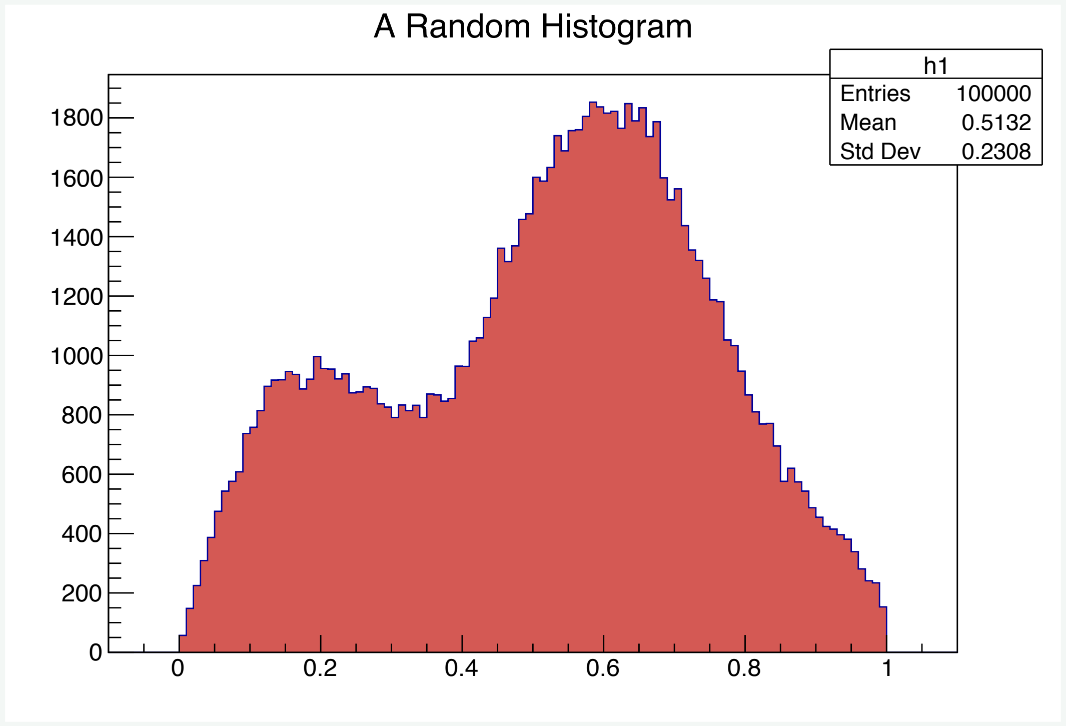
THE HIT-OR-MISS METHOD (II)

- ➤ Such an idea can be implemented quickly (nothing really difficult in fact!)

example_04.cc

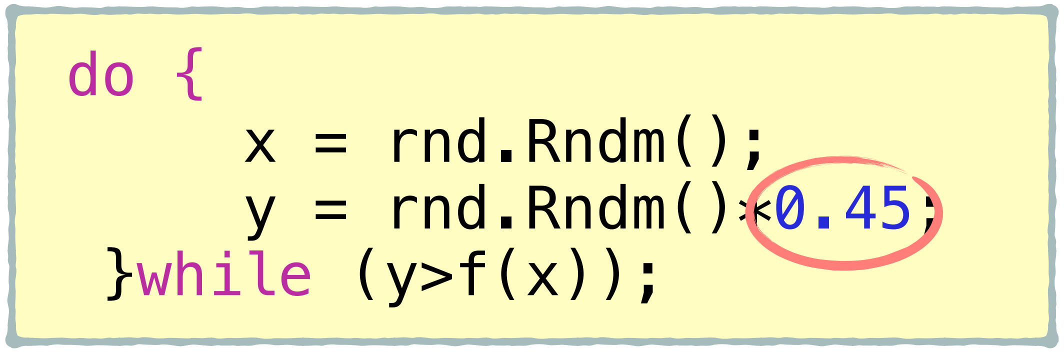
```
double f(double x)
{  return x - x*x + x*x*x - x*x*x*x + sin(x*13.)/13.; }

void example_04()
{
    TRandom3 rnd;
    TH1F *h1 = new TH1F("h1","A Random Histogram",120,-0.1,1.1);

    for(int i=0;i<100000;i++) {
        double x,y;
        do {
            x = rnd.Rndm();
            y = rnd.Rndm()*0.45;
        }while (y>f(x));

        h1->Fill(x);
    }

    h1->SetFillColor(50);
    h1->Draw();
}
```

generation in 2D



THE STRANGE SCALE?

- ➤ You may notice there is a strange(?) **0.45** scaling factor in the code. This is due to the fact that the given function f(x) is always smaller than 0.45 in the given range of generation:



```
do {
      x = rnd.Rndm();
      y = rnd.Rndm()*0.45;
 }while (y>f(x));
```

- ➤ What happen if we do not know about it? In principle one can do a scan first, but it's not very efficient for higher dimensional functions.
- ➤ The key idea: **importance sampling** – we could just use few random numbers to find the maximum value of the function.

WHAT IF YOU DON'T KNOW ABOUT THE UPPER BOUND?

- One can just "waste" some random generation calls and estimate the upper bound first. And use this upper bound to do the real generation.

example_04a.cc

```
double f_max = 0.;
 for(int i=0;i<1000;i++) {
     double y = f(rnd.Rndm());
     if (y>f_max) f_max = y;
 }
 f_max *= 1.2;

 for(int i=0;i<100000;i++) {
     double x,y;
     do {
         x = rnd.Rndm();
         y = rnd.Rndm()*f_max;
     }while (y>f(x));

     h1->Fill(x);
 }
```

f_max ~ 0.40336

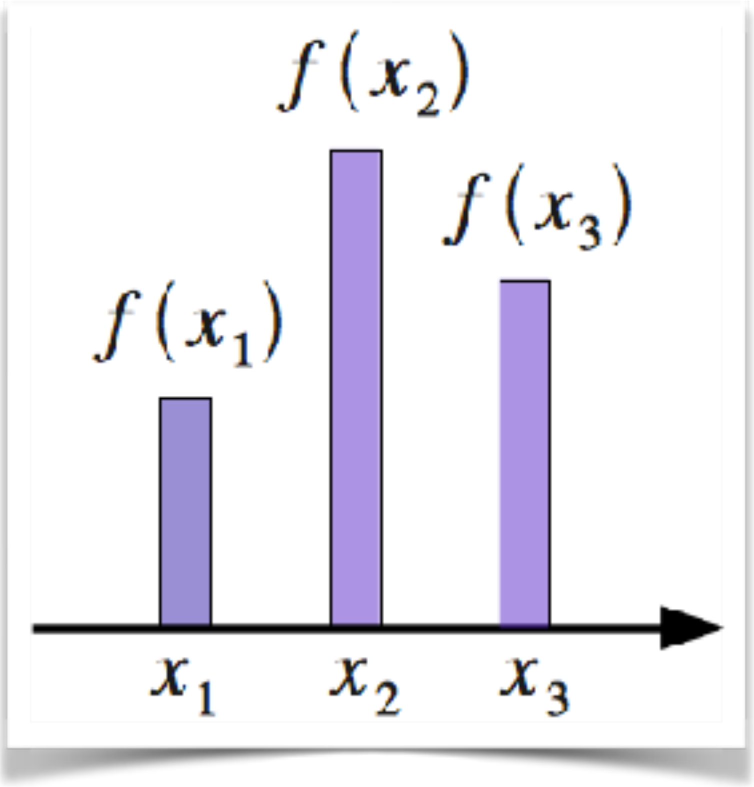
add 20% spare space!

one can actually add a protection here, e.g.:

```
if (f(x) > f_max) {
    //print out something...
    exit(1);
}
```

IT'S NOT QUITE EFFICIENT, RIGHT?

- We already know that the (*pseudo*) random numbers are basically limited: **limited period** & **limited computing time**.
- In principle one could use a much more efficient way to generate the random distributions: **Inverse Transform Method**.
- Consider a 3-bin function (*as an example*):



Generate **r∈[0,1]** uniformly:

$$\text{if } r < \frac{f(x_1)}{f(x_1)+f(x_2)+f(x_3)} \text{ then } x = x_1 \text{ else}$$

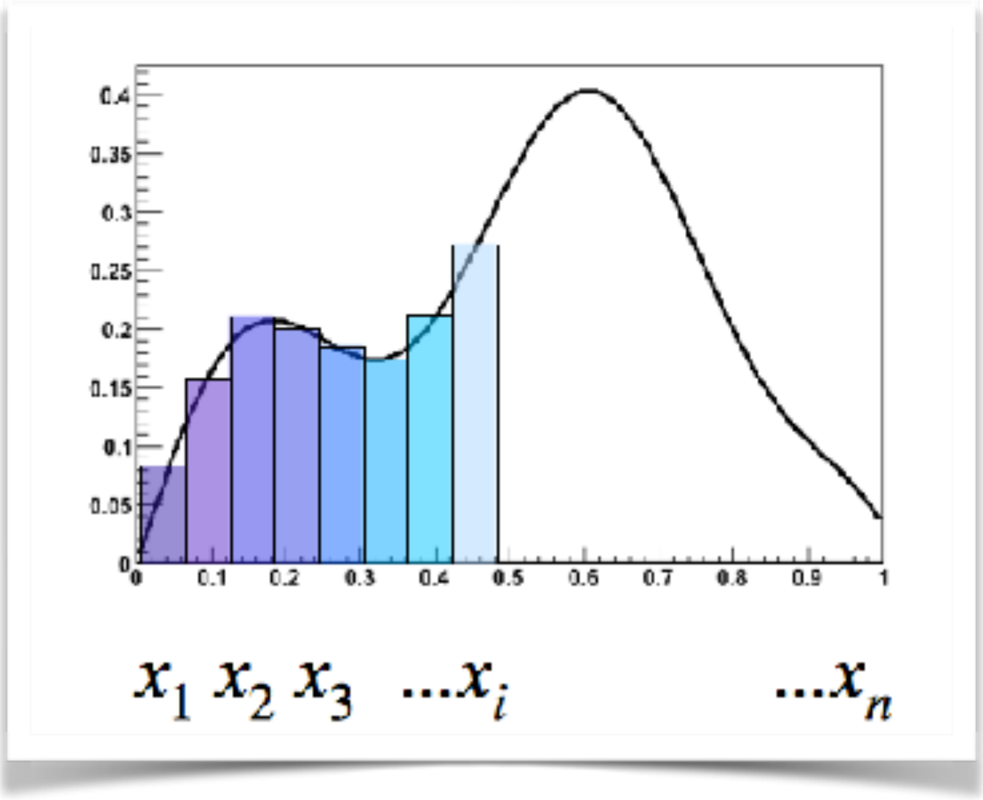
$$\text{if } r < \frac{f(x_1)+f(x_2)}{f(x_1)+f(x_2)+f(x_3)} \text{ then } x = x_2 \text{ else}$$

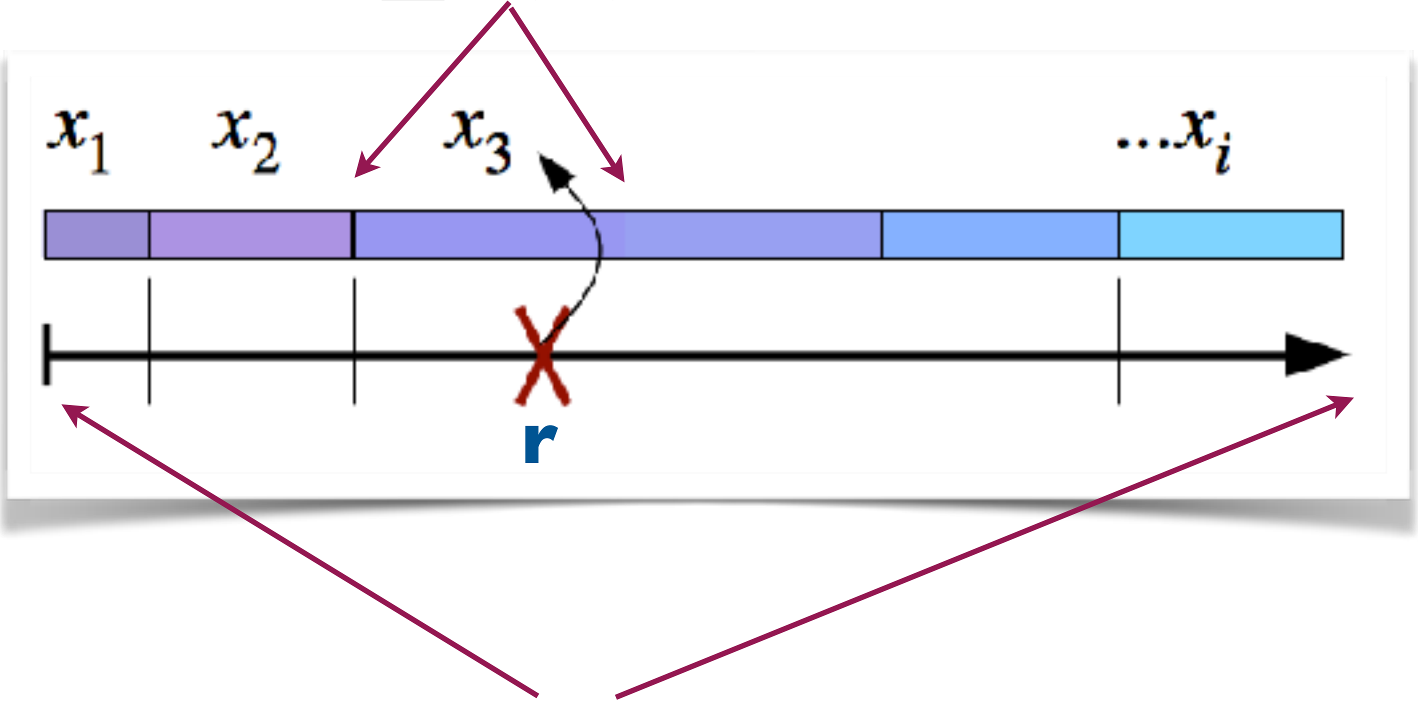
$$\text{if } r < \frac{f(x_1)+f(x_2)+f(x_3)}{f(x_1)+f(x_2)+f(x_3)} \text{ then } x = x_3$$

IT'S NOT QUITE EFFICIENT, RIGHT? (II)

- For a multi-bin case:

$$\frac{f(x_3)}{\sum f(x_i)}$$





Generate **r**∈**[0,1]** uniformly, and inverse transform back to x by

$$\text{if } \frac{f(x_1)+f(x_2)+\ldots+f(x_{m-1})}{\sum f(x_i)} \leq r < \frac{f(x_1)+f(x_2)+\ldots+f(x_{m-1})+f(x_m)}{\sum f(x_i)}$$

then take x = x_m

IT'S NOT QUITE EFFICIENT, RIGHT? (III)

- For more and more bins:

$$\text{if } \frac{f(x_1)+f(x_2)+\ldots+f(x_{m-1})}{\sum f(x_i)} \leq r < \frac{f(x_1)+f(x_2)+\ldots+f(x_{m-1})+f(x_m)}{\sum f(x_i)} \text{ then } x = x_m$$

$$\longrightarrow \text{if } \frac{\int_a^{x_m} f(x')dx}{\int_a^b f(x')dx'} \leq r < \frac{\int_a^{x_m+\delta} f(x')dx}{\int_a^b f(x')dx'} \text{ then } x = x_m$$

$$\longrightarrow \text{if } \frac{\int_a^{x_m} f(x')dx}{\int_a^b f(x')dx'} = r \text{ then } x = x_m$$

Given **r**∈**[0,1]** and
solve the equation to obtain **x**.

IN EXPLICIT MATHEMATICS

$$W(x) = \frac{\int_a^x f(x')dx'}{\int_a^b f(x')dx'} = r \text{ (a random number in } [0,1])$$

Find the invert function of $W(x) \to x = W^{-1}(r)$

For example: $f(x) = \exp(-x);\ [a,b] = [0,1]$

$$\int_0^x \exp(-x')dx' = 1 - \exp(-x)$$

$$W(x) = \frac{\int_0^x \exp(-x')dx'}{\int_0^1 \exp(-x')dx'} = \frac{1-\exp(-x)}{1-\exp(-1)}$$

$$x = W^{-1}(r) = -\log\left(1 - r + \frac{r}{e}\right)$$

Generate **r**∈**[0,1]** then convert to **x**.

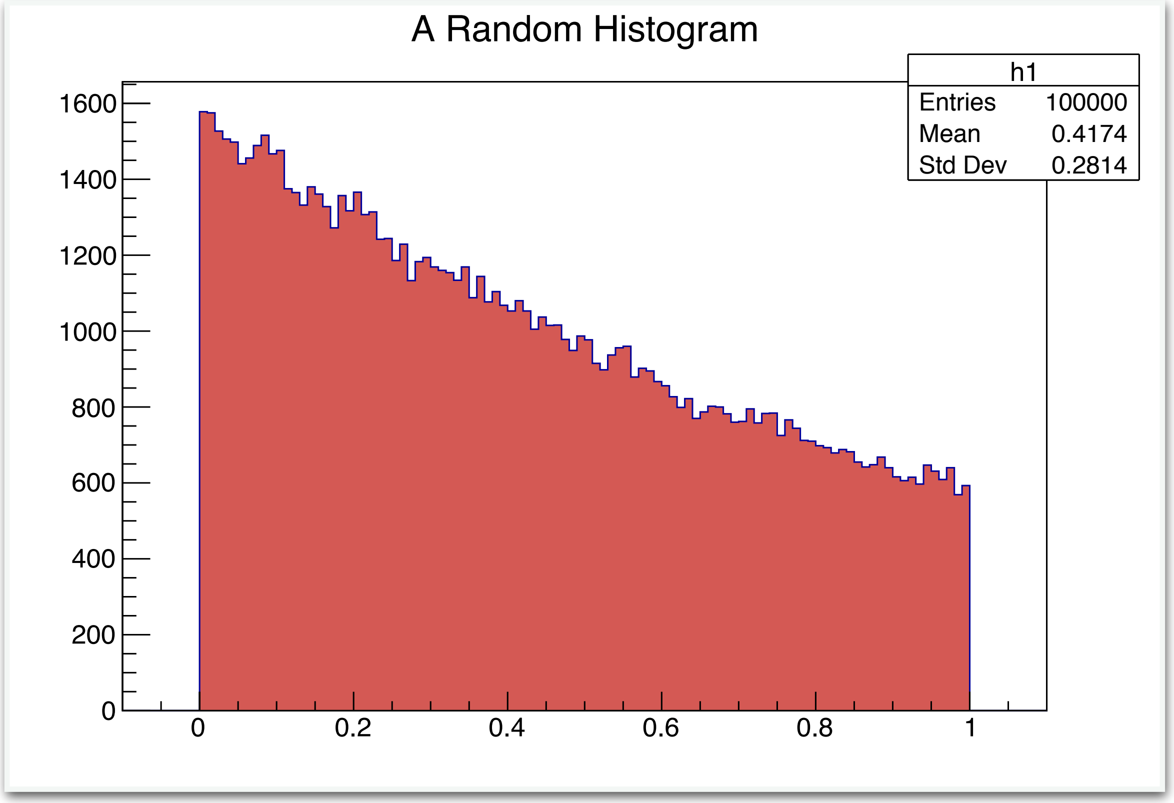
LET'S GIVE IT A TRY!

➤ Given the explicit formula given for the distribution below:

$$x = W^{-1}(r) = -\log\left(1 - r + \frac{r}{e}\right)$$

Generate **r∈[0,1]** then convert to **x**.

the generation is very straightforward:



example_05.cc

```
{
    TRandom3 rnd;

    TH1F *h1 = new TH1F("h1",
"A Random Histogram",120,-0.1,1.1);

    for(int i=0;i<100000;i++) {
        double r = rnd.Rndm();
        double x = -log(1.-r+r/M_E);

        h1->Fill(x);
    }

    h1->SetFillColor(50);
    h1->Draw();
}
```

EXTENDING TO INFINITY

- In the case of unbounded distribution – if the upper bound is infinity – this cannot be carried out by a hit-or-miss method! e.g.

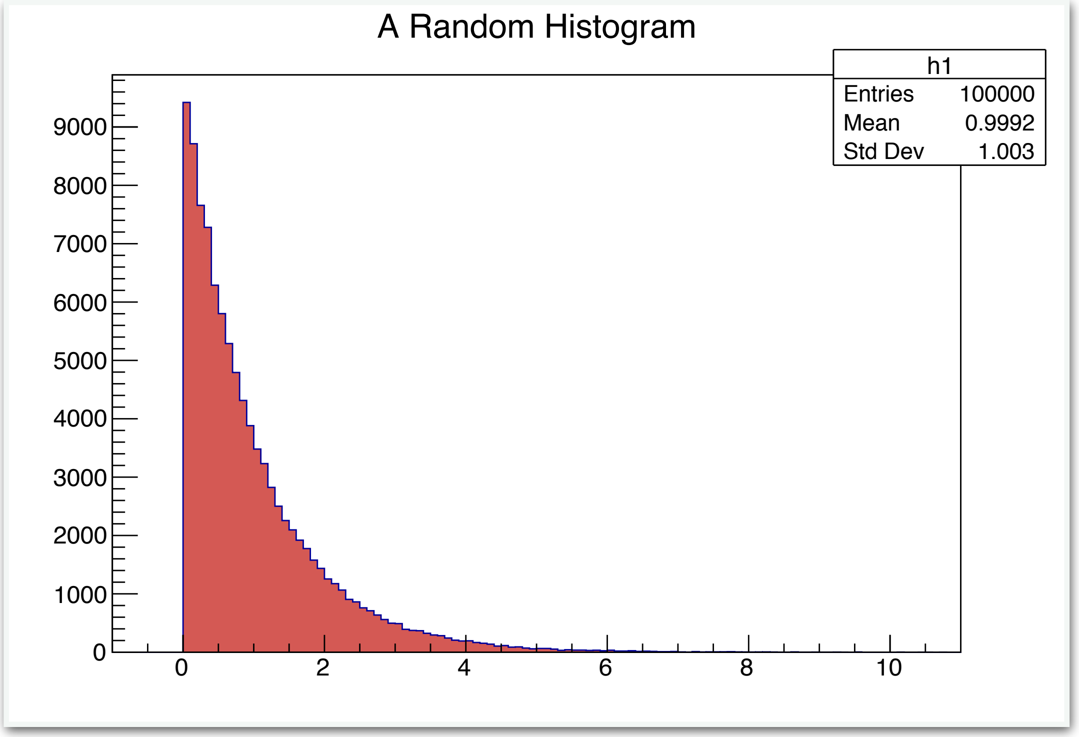
$$f(x) = \exp(-x);\ \ [a,b] = [0, \infty]$$

$$W(x) = \frac{\int_0^x \exp(-x')dx'}{\int_0^\infty \exp(-x')dx'} = 1 - \exp(-x)$$

$$\longrightarrow\ x = W^{-1}(r) = -\log(1-r)$$

example_05a.cc

```
for(int i=0;i<100000;i++) {
    double r = rnd.Rndm();
    double x = -log(1.-r);

    h1->Fill(x);
 }
```



GENERATE A GAUSSIAN

- Instead of hit-or-mass method, let's practice the inverse transformation with the function form:

$$G(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left[\frac{-(x-\mu)^2}{2\sigma^2}\right]$$

$$\text{For } \mu = 0, \sigma = 1 \;\rightarrow\; G(x) = \frac{1}{\sqrt{2\pi}}\exp\left[\frac{-(x)^2}{2}\right]$$

$$I^2 = \iint G(x)G(y)dxdy = \iint \frac{1}{2\pi}\exp\left(\frac{-x^2+y^2}{2}\right)dxdy = \iint \frac{1}{2\pi}\exp\left(\frac{-r^2}{2}\right)rd\phi dr$$

$$I^2 = \left[-\exp\left(\frac{-r^2}{2}\right)\right]_0^R \times \left[\frac{\phi}{2\pi}\right]_0^\Phi \quad \text{for } R = \infty, \Phi = 2\pi \;\rightarrow\; I^2 = 1$$

assign random number r_1

assign random number r_2

GENERATE A GAUSSIAN (II)

$$I^2 = \left[-\exp\left(\frac{-r^2}{2}\right)\right]_0^R \times \left[\frac{\phi}{2\pi}\right]_0^\Phi$$

$$\rightarrow \quad 1 - \exp\left(\frac{-R^2}{2}\right) = r_1 \rightarrow R = \sqrt{-2\log(1-r_1)}$$

$$\rightarrow \quad \frac{\Phi}{2\pi} = r_2 \rightarrow \Phi = 2\pi r_2$$

- Change the variables back to **x,y**:

$$x = R\cos\Phi = \sqrt{-2\log(1-r_1)}\cos(2\pi r_2)$$
$$y = R\sin\Phi = \sqrt{-2\log(1-r_1)}\sin(2\pi r_2)$$

or

$$x = \sqrt{-2\log(r_1)}\cos(2\pi r_2)$$
$$y = \sqrt{-2\log(r_1)}\sin(2\pi r_2)$$

*Since it does not matter if we generate **r₁** or **1-r₁***

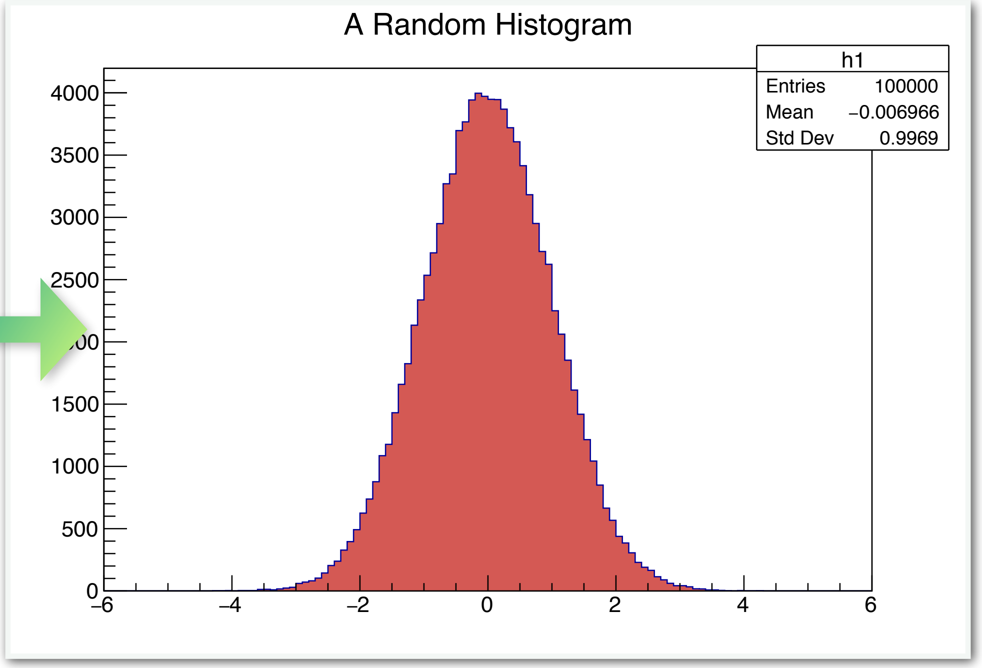
Both x and y should be good Gaussian random variables (and they are independent)!

GENERATE A GAUSSIAN (III)

- The generation is very straightforward again:

example_06.cc

```
for(int i=0;i<100000;i++) {
  double r1 = rnd.Rndm();
  double r2 = rnd.Rndm();
  double x =
      sqrt(-2.*log(r1))*cos(2.*M_PI*r2);

  h1->Fill(x);
}
```

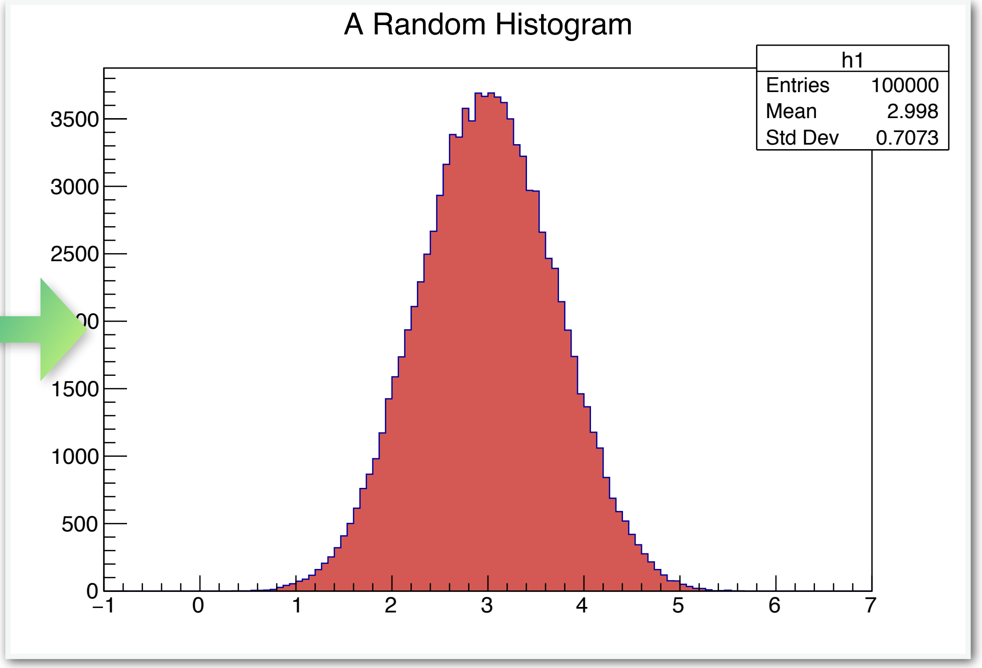


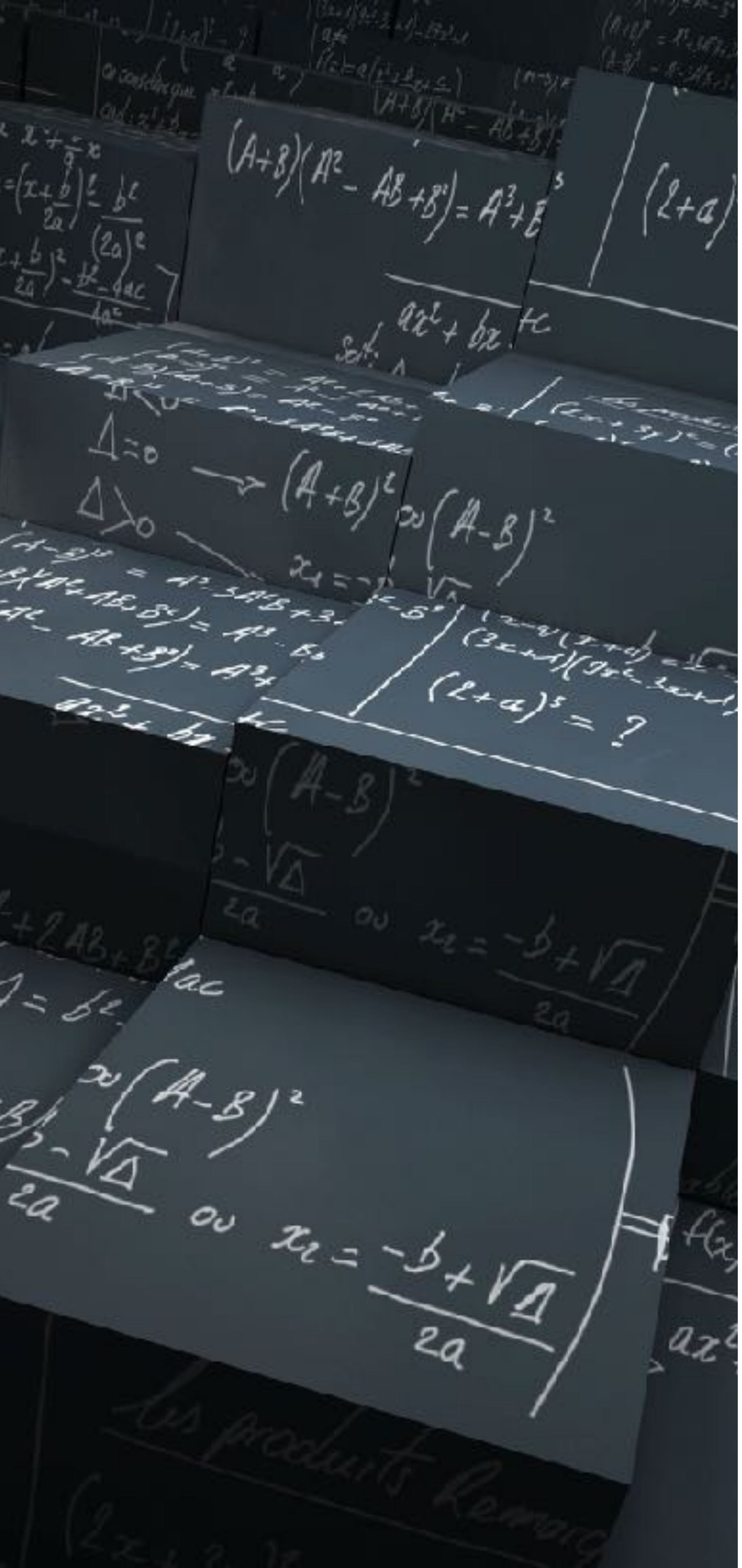
- It might be cute(?) to try this:

example_06a.cc

```
for(int i=0;i<100000;i++) {
  double x = rnd.Rndm()+rnd.Rndm()+rnd.Rndm()+
             rnd.Rndm()+rnd.Rndm()+rnd.Rndm();

  h1->Fill(x);
}
```

This is the central limit theorem.
Will be discussed in the next lecture!





SUMMARY

- In this lecture we discussed several important aspects regarding random numbers –– the properties of the random number generators in the ROOT framework, and how to generate non-uniform distributed distributions from a uniform distribution!
- This should cover the minimal needs of random number related topics for the following lectures and exercises!