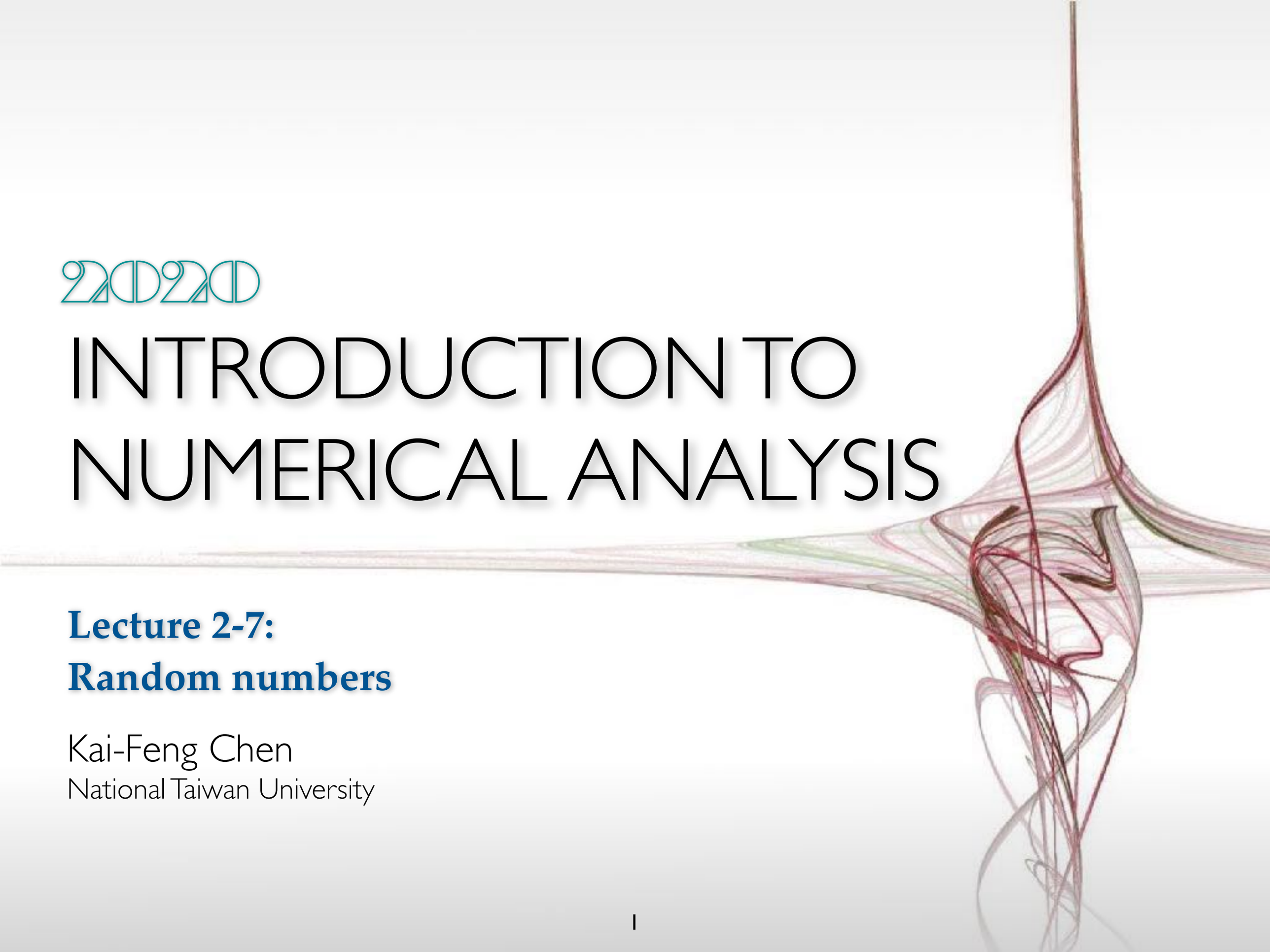
2020

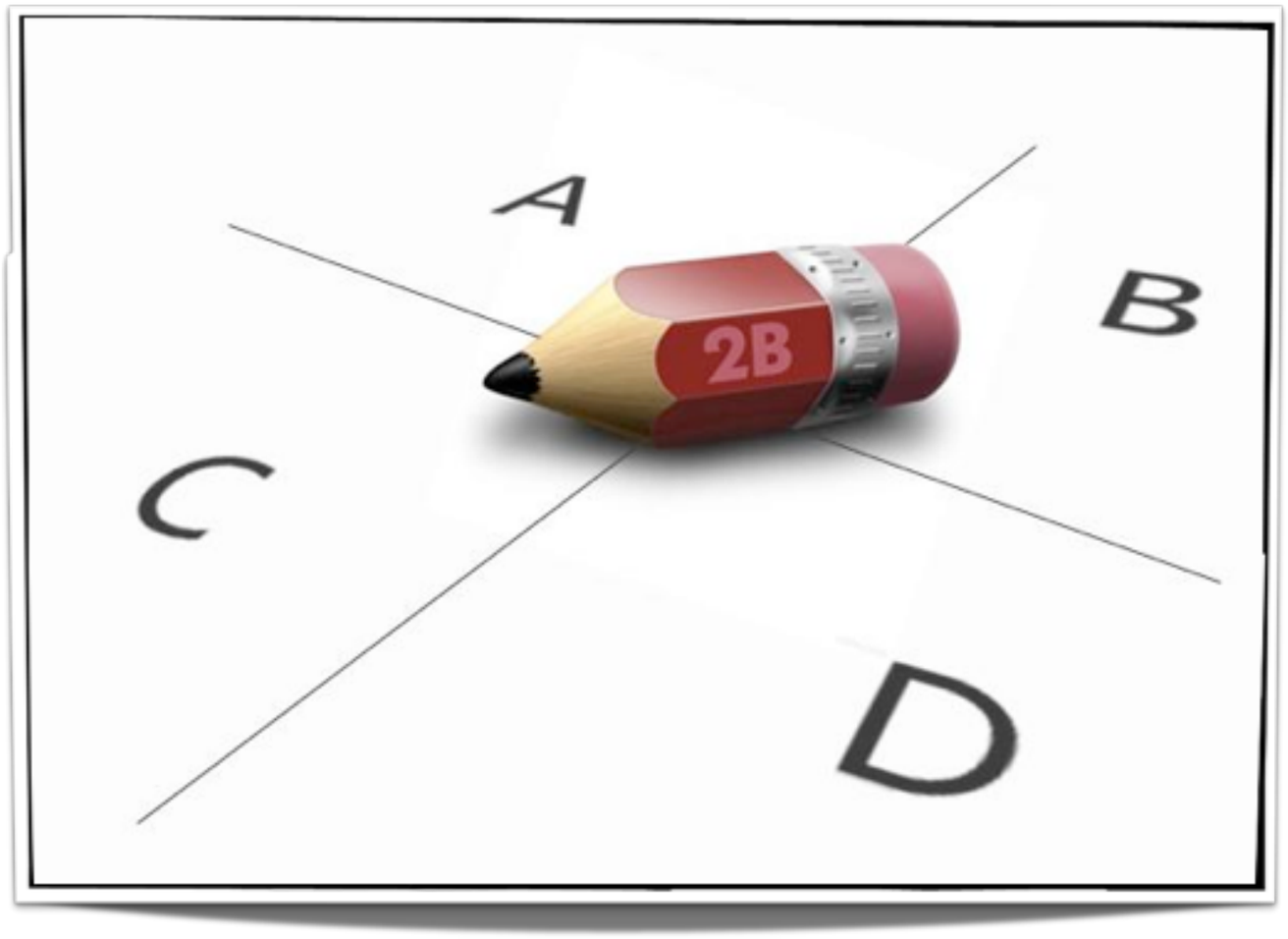
INTRODUCTION TO NUMERICAL ANALYSIS



Lecture 2-7: Random numbers

Kai-Feng Chen
National Taiwan University

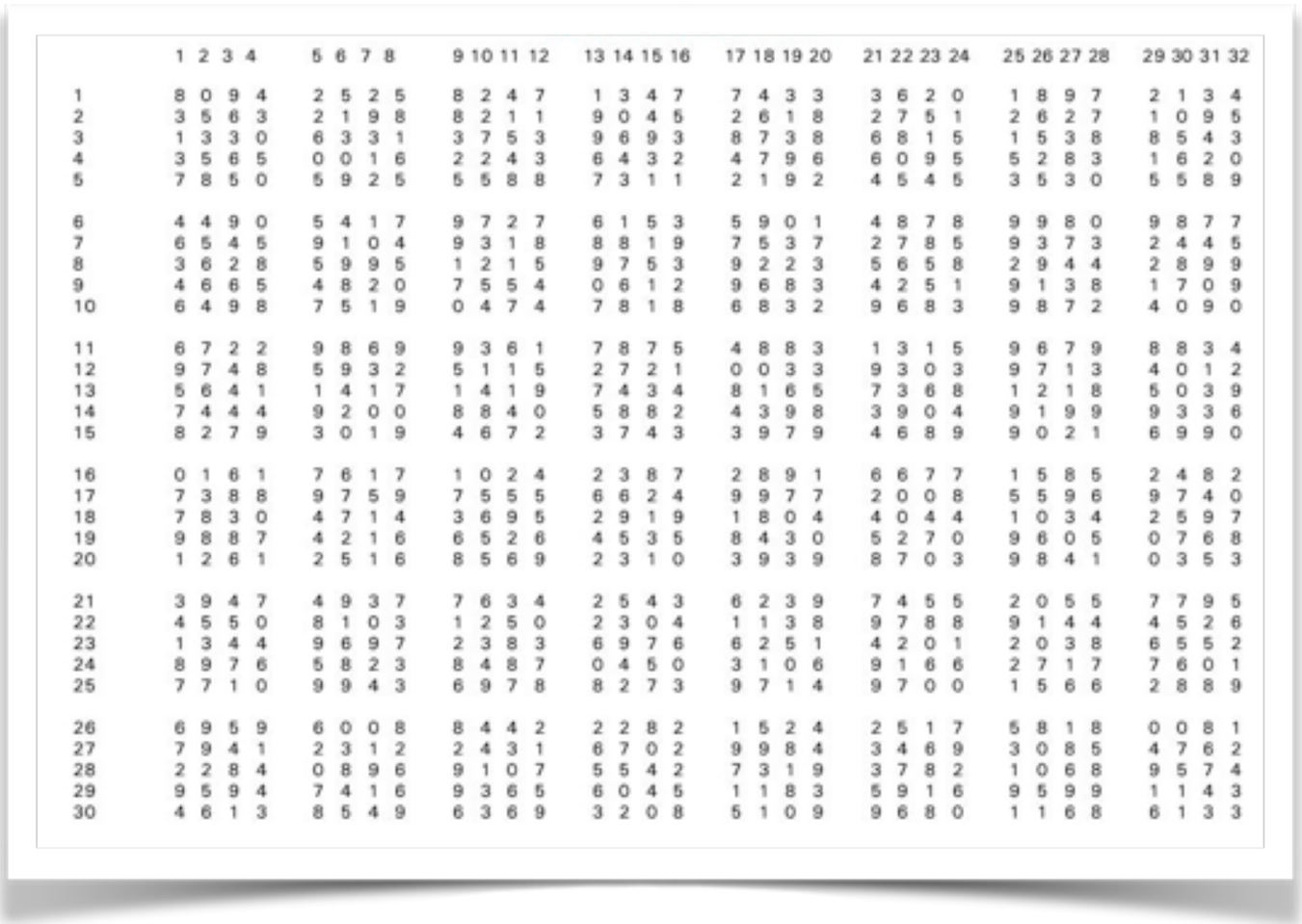
OLD-FASHIONED RANDOM NUMBERS



Well, this is still very useful in some special situation...

But this old tech random number generator cannot generate too many digits, and it may not be fair enough.

AN OLD-FASHIONED METHOD: RANDOM TABLE



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	8	0	9	4	2	5	2	5	8	2	4	7	1	3	4	7	7	4	3	3	3	6	2	0	1	8	9	7	2	1	3	4
2	3	5	6	3	2	1	9	8	8	2	1	1	9	0	4	5	2	6	1	8	2	7	5	1	2	6	2	7	1	0	9	5
3	1	3	3	0	6	3	3	1	3	7	5	3	9	6	9	3	8	7	3	8	6	8	1	5	1	5	3	8	8	5	4	3
4	3	5	6	5	0	0	1	6	2	2	4	3	6	4	3	2	4	7	9	6	6	0	9	5	5	2	8	3	1	6	2	0
5	7	8	5	0	5	9	2	5	5	5	8	8	7	3	1	1	2	1	9	2	4	5	4	5	3	5	3	0	5	5	8	9
6	4	4	9	0	5	4	1	7	9	7	2	7	6	1	5	3	5	9	0	1	4	8	7	8	9	9	8	0	9	8	7	7
7	6	5	4	5	9	1	0	4	9	3	1	8	8	8	1	9	7	5	3	7	2	7	8	5	9	3	7	3	2	4	4	5
8	3	6	2	8	5	9	9	5	1	2	1	5	9	7	5	3	9	2	2	3	5	6	5	8	2	9	4	4	2	8	9	9
9	4	6	6	5	4	8	2	0	7	5	5	4	0	6	1	2	9	6	8	3	4	2	5	1	9	1	3	8	1	7	0	9
10	6	4	9	8	7	5	1	9	0	4	7	4	7	8	1	8	6	8	3	2	9	6	8	3	9	8	7	2	4	0	9	0
11	6	7	2	2	9	8	6	9	9	3	6	1	7	8	7	5	4	8	8	3	1	3	1	5	9	6	7	9	8	8	3	4
12	9	7	4	8	5	9	3	2	5	1	1	5	2	7	2	1	0	0	3	3	9	3	0	3	9	7	1	3	4	0	1	2
13	5	6	4	1	1	4	1	7	1	4	1	9	7	4	3	4	8	1	6	5	7	3	6	8	1	2	1	8	5	0	3	9
14	7	4	4	4	9	2	0	0	8	8	4	0	5	8	8	2	4	3	9	8	3	9	0	4	9	1	9	9	9	3	3	6
15	8	2	7	9	3	0	1	9	4	6	7	2	3	7	4	3	3	9	7	9	4	6	8	9	9	0	2	1	6	9	9	0
16	0	1	6	1	7	6	1	7	1	0	2	4	2	3	8	7	2	8	9	1	6	6	7	7	1	5	8	5	2	4	8	2
17	7	3	8	8	9	7	5	9	7	5	5	5	6	6	2	4	9	9	7	7	2	0	0	8	5	5	9	6	9	7	4	0
18	7	8	3	0	4	7	1	4	3	6	9	5	2	9	1	9	1	8	0	4	4	0	4	4	1	0	3	4	2	5	9	7
19	9	8	8	7	4	2	1	6	6	5	2	6	4	5	3	5	8	4	3	0	5	2	7	0	9	6	0	5	0	7	6	8
20	1	2	6	1	2	5	1	6	8	5	6	9	2	3	1	0	3	9	3	9	8	7	0	3	9	8	4	1	0	3	5	3
21	3	9	4	7	4	9	3	7	7	6	3	4	2	5	4	3	6	2	3	9	7	4	5	5	2	0	5	5	7	7	9	5
22	4	5	5	0	8	1	0	3	1	2	5	0	2	3	0	4	1	1	3	8	9	7	8	8	9	1	4	4	4	5	2	6
23	1	3	4	4	9	6	9	7	2	3	8	3	6	9	7	6	6	2	5	1	4	2	0	1	2	0	3	8	6	5	5	2
24	8	9	7	6	5	8	2	3	8	4	8	7	0	4	5	0	3	1	0	6	9	1	6	6	2	7	1	7	7	6	0	1
25	7	7	1	0	9	9	4	3	6	9	7	8	8	2	7	3	9	7	1	4	9	7	0	0	1	5	6	6	2	8	8	9
26	6	9	5	9	6	0	0	8	8	4	4	2	2	2	8	2	1	5	2	4	2	5	1	7	5	8	1	8	0	0	8	1
27	7	9	4	1	2	3	1	2	2	4	3	1	6	7	0	2	9	9	8	4	3	4	6	9	3	0	8	5	4	7	6	2
28	2	2	8	4	0	8	9	6	9	1	0	7	5	5	4	2	7	3	1	9	3	7	8	2	1	0	6	8	9	5	7	4
29	9	5	9	4	7	4	1	6	9	3	6	5	6	0	4	5	1	1	8	3	5	9	1	6	9	5	9	9	1	1	4	3
30	4	6	1	3	8	5	4	9	6	3	6	9	3	2	0	8	5	1	0	9	9	6	8	0	1	1	6	8	6	1	3	3

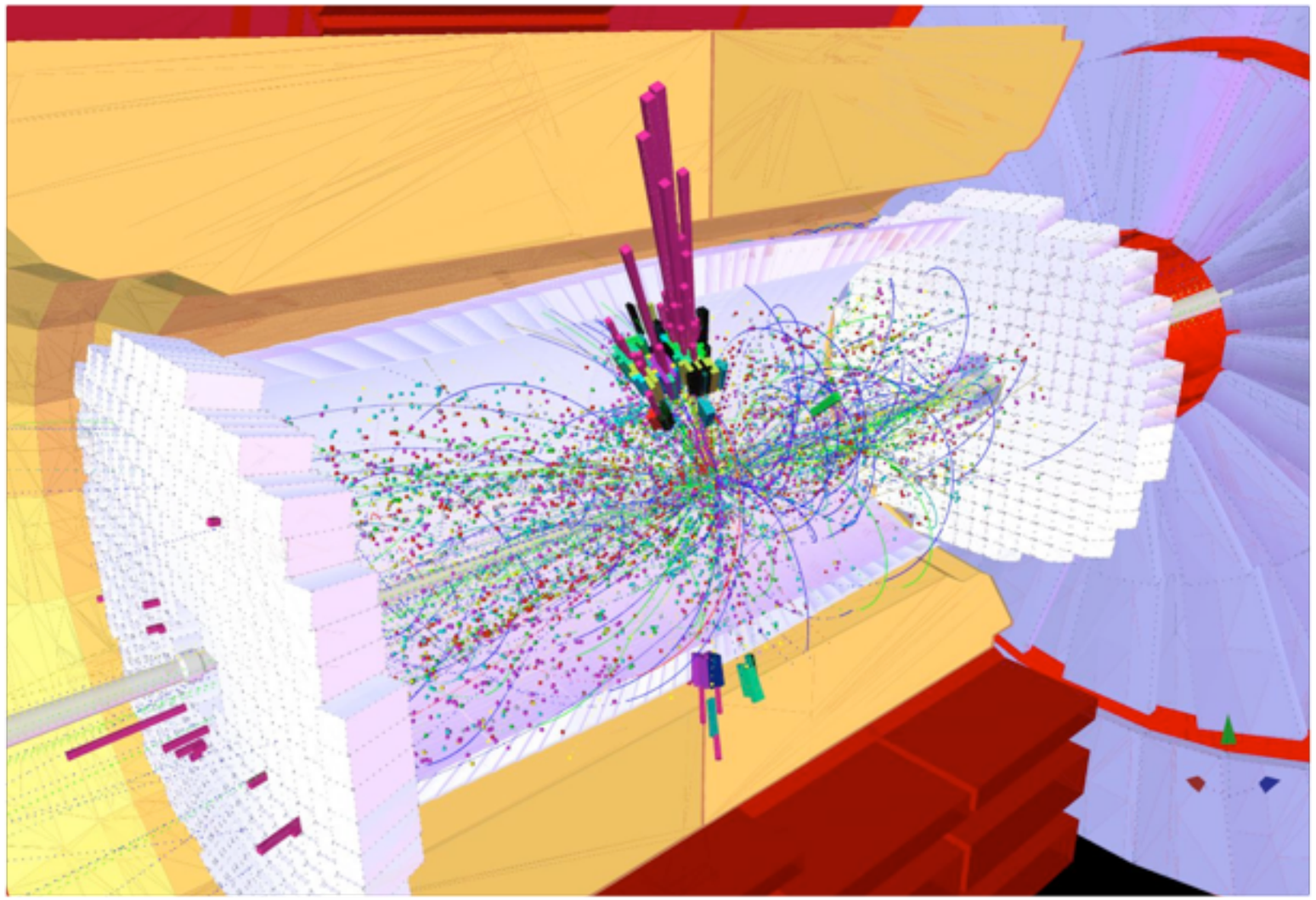
Have you generate some random numbers using a **random table** at high school?

This actually shows two main ideas of pseudo random numbers –

1) You need a seed to start
2) recursive operation

Let's do a similar job with your computer!

A (MORE) MODERN APPLICATION



Simulation of a not-yet-observed particle named Z' decaying into high momentum quarks.

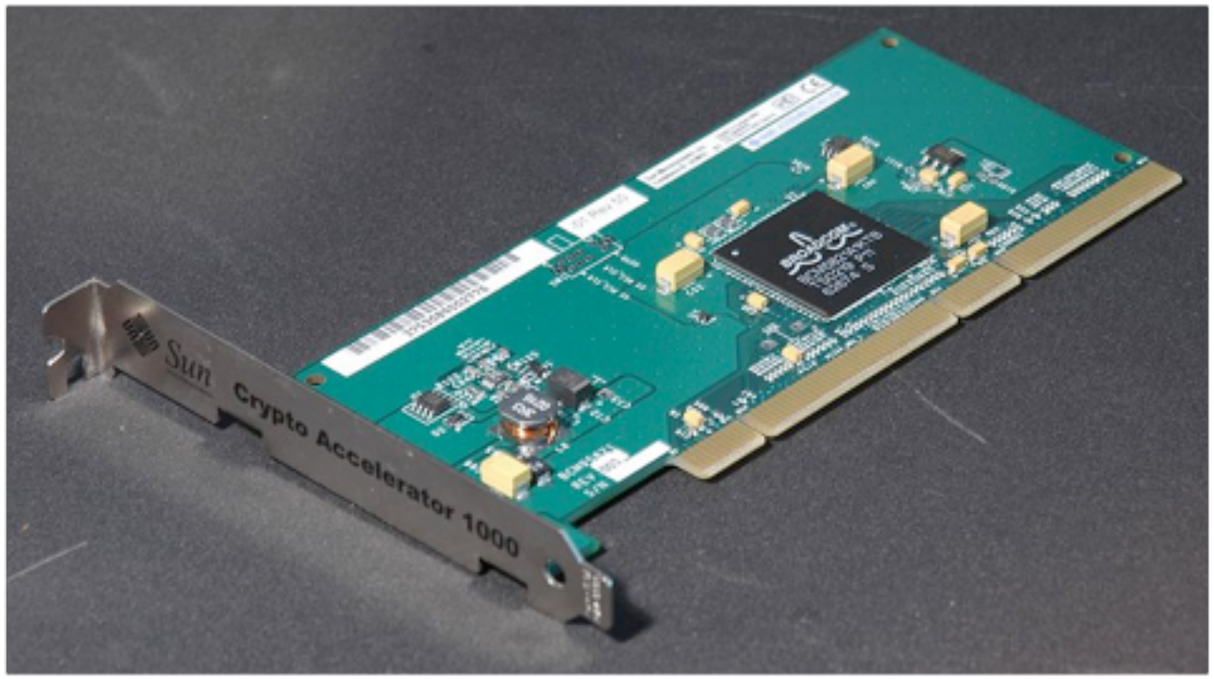
"TRUE" VERSUS "PSEUDO" GENERATORS

- 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.

IN COMPUTING: A CLASSICAL ALGORITHM

- A classical, commonly used everywhere, but it's actually a bad algorithm: **Linear Congruential Generator (LCG)** –

$$R' = [a \cdot R + c] \mod M$$

***R**: current value (Seed)*
***R'**: next value*

- **a,c**: scaling constants
- **M**: the module number
- **Period**: **O(<M)** for some good **a** and **c**

For some certain "good" values of a and c, we could produce close-to (pseudo) random numbers by computers easily.

However a "bad" selection of a,c values will give a terrible result!

JUST TRY IT OUT

```
class rnd:
    def __init__(self, s = 1234):
        self.seed = s

    def gen(self):
        self.seed = (32533521*self.seed + 2424) % 100
        return self.seed

r = rnd()
for i in range(30):
    print(r.gen(), end=' ')
```

l207-example-01.py

```
First few numbers
38, 22, 86, 30, 54, 58, 42, 6, 50, 74, 78, 62, 26, 70, 94, 98,
82, 46, 90, 14, 18, 2, 66, 10, 34, 38, 22, 86, 30, 54, ...
```

Started to repeat the same numbers;
the actual **period** is smaller than the value of M (=100 here).

SELECTING "GOOD" COEFFICIENTS

- Actually it's a kind of **ART** to find good coefficients:

```
For a = 9289, c = 4, M = 100:
30, 74, 90, 14, 50, 54, 10, 94, 70, 34, 30, 74, 90, 14,
50, 54, 10, 94, 70, 34, 30, 74, 90, 14, 50, 54, 10, 94,
70, 34, ...
```

```
For a = 928983621, c = 1286825, M = 100:
39, 44, 49, 54, 59, 64, 69, 74, 79, 84, 89, 94, 99, 4,
9, 14, 19, 24, 29, 34, 39, 44, 49, 54, 59, 64, 69, 74,
79, 84, ...
```

```
For a = 77777, c = 99999, M = 100:
17, 8, 15, 54, 57, 88, 75, 74, 97, 68, 35, 94, 37, 48,
95, 14, 77, 28, 55, 34, 17, 8, 15, 54, 57, 88, 75, 74,
97, 68, ...
```

SELECTING "GOOD" COEFFICIENTS (II)

- Some widely used selection of constants (source: Wikipedia), you can see this type of generator is actually used everywhere!

Source	m	a	c	output bits of seed in *rand()* / *Random(L)*
Numerical Recipes	2^{32}	1664525	1013904223	
Borland C/C++	2^{32}	22695477	1	bits 30..16 in *rand()*, 30..0 in *lrand()*
glibc (used by GCC)[4]	2^{32}	1103515245	12345	bits 30..0
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++	2^{32}	1103515245	12345	bits 30..16
Borland Delphi, Virtual Pascal	2^{32}	134775813	1	bits 63..32 of *(seed * L)*
Microsoft Visual/Quick C/C++	2^{32}	214013	2531011	bits 30..16
Apple CarbonLib	$2^{31} - 1$	16807	0	see Park–Miller RNG
MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407	
VAX's **MTH$RANDOM**,[5] old versions of glibc	2^{32}	69069	1	
Random class in Java API	2^{48}	25214903917	11	bits 48...17

A WELL-KNOWN BAD SELECTION

- A routine, **RANDU**, use the selection of **a=65549, c=0, M=2^{31}**.
- This was widely used at many places during 1960s by IBM.
- Quotations:

  …its very name RANDU is enough to bring dismay into the eyes and stomachs of many computer scientists!

  —Donald Knuth

  One of us recalls producing a "random" plot with only 11 planes, and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random."

  —W. H. Press et al.

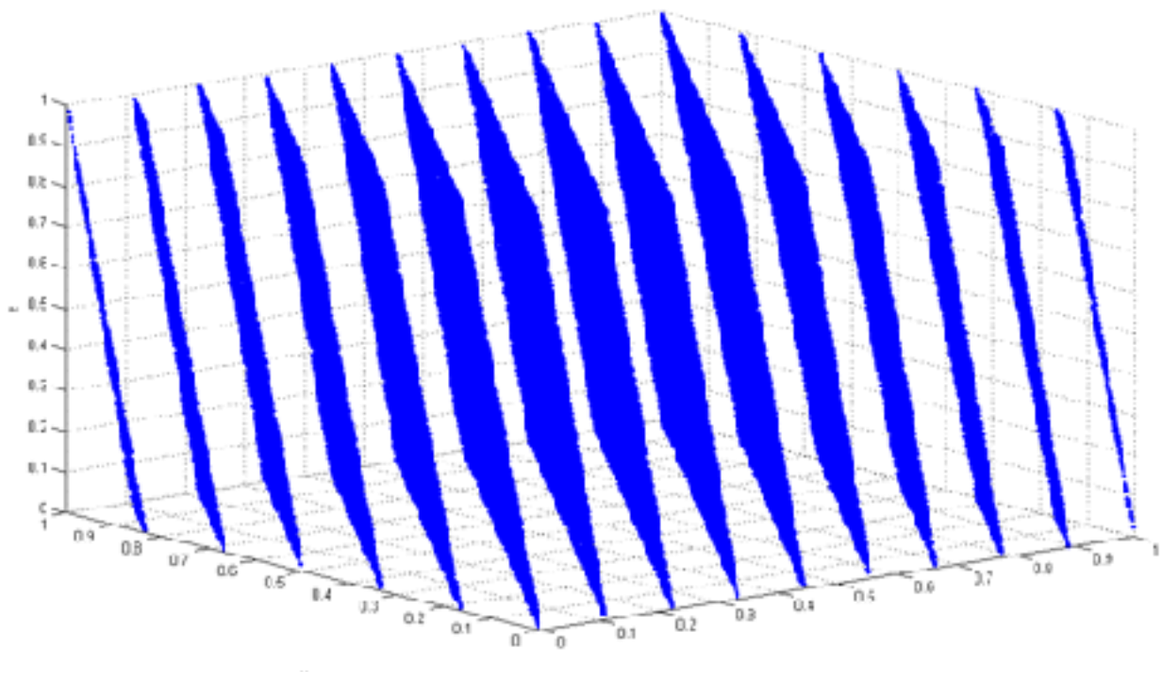
- First few numbers (seed = 1):
  65539, 393225, 1769499, 7077969, 26542323, 95552217, ...

A WELL-KNOWN BAD SELECTION (II)

- The period length of RANDU is 536,870,901 if seed = 1 (it's already a large number, isn't it?)
- It can generate an uniform distribution between (0,1).
- The real problem: if one do the math carefully, the following relation can be obtained:

$$R_{i+2} = 6R_{i+1} - 9R_i$$

- If one draw R_i **versus** R_{i+1} **versus** R_{i+2} in a 3D plot, one can observe a funny correlation:



ONE GOOD WORKING EXAMPLE: PARK-MILLER RNG

- Use a = **16807**, c = **0**, M = **2^{31}–1**.
- The period can reach **2,147,483,645** if seed = 1234 (it's almost full 2^{31}–1!)

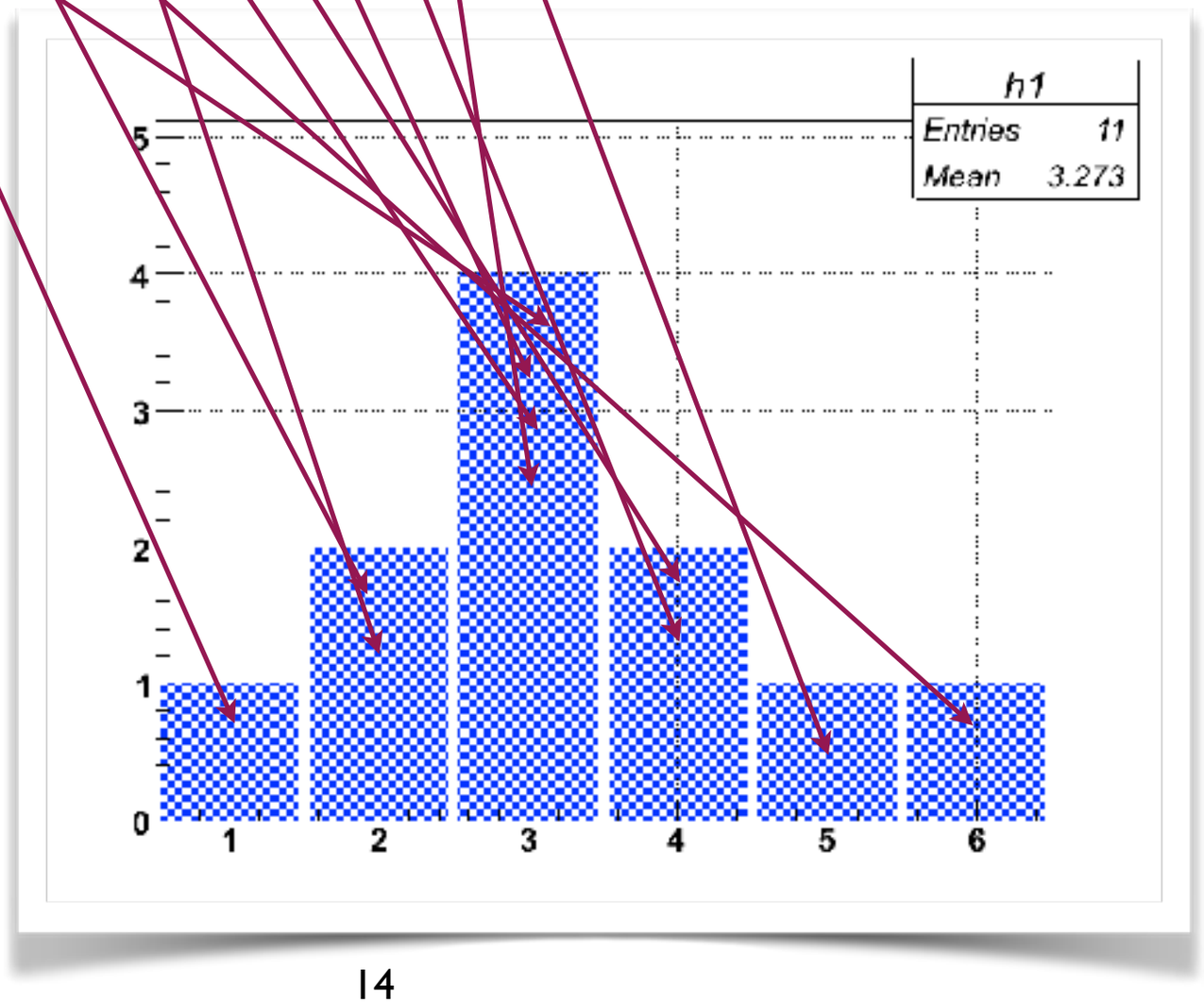
```
class rnd:
    def __init__(self, s = 1234):
        self.seed = s

    def gen(self):
        self.seed = (16807*self.seed) % (2**31-1)
        return self.seed
```

l207-example-01a.py (partial)

```
First few numbers
20739838, 682106452, 895431078, 2092213417, 933663541, 420124958,
113937770, 1544170913, 540660796, 882687915, 518753929,
2061161530, 883124953, 1421600654, 2086618903, ...
```

DISTRIBUTION AS A HISTOGRAM

- Histogram is just occurrence counting, i.e. how often they appear;
  For example: **{1,3,2,6,2,3,4,3,4,3,5}**

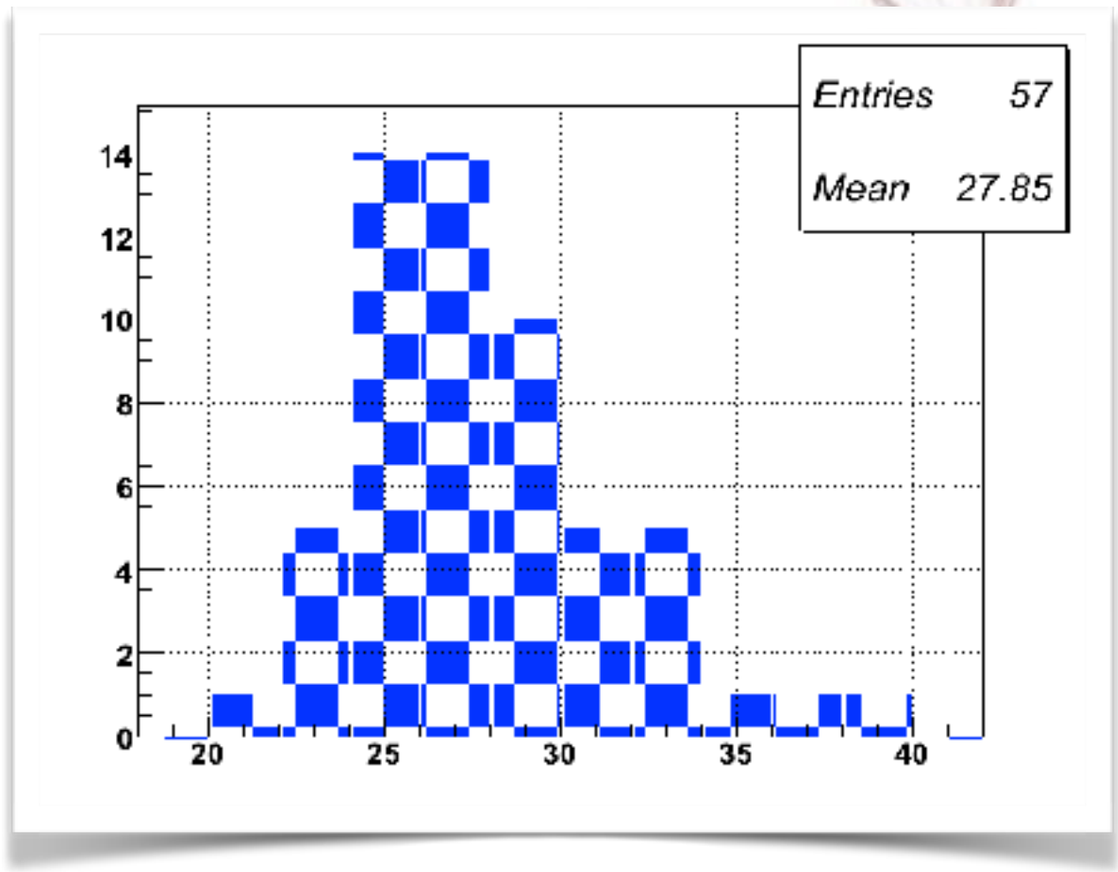


DISTRIBUTION AS A HISTOGRAM (II)

- How is a histogram made? Lets consider an age distribution as following:

Ages
26.93
23.03
29.66
33.74
32.19
31.40
24.88
27.56
26.99
26.65

Age	Number
20-22	1
22-24	5
24-26	14
26-28	14
28-30	10
30-32	5
32-34	5
34-36	1
36-38	1
38-40	1



This is a usual way to show the **"probability of happening"** in a specific **"interval"**.

PLOTTING THE DISTRIBUTION

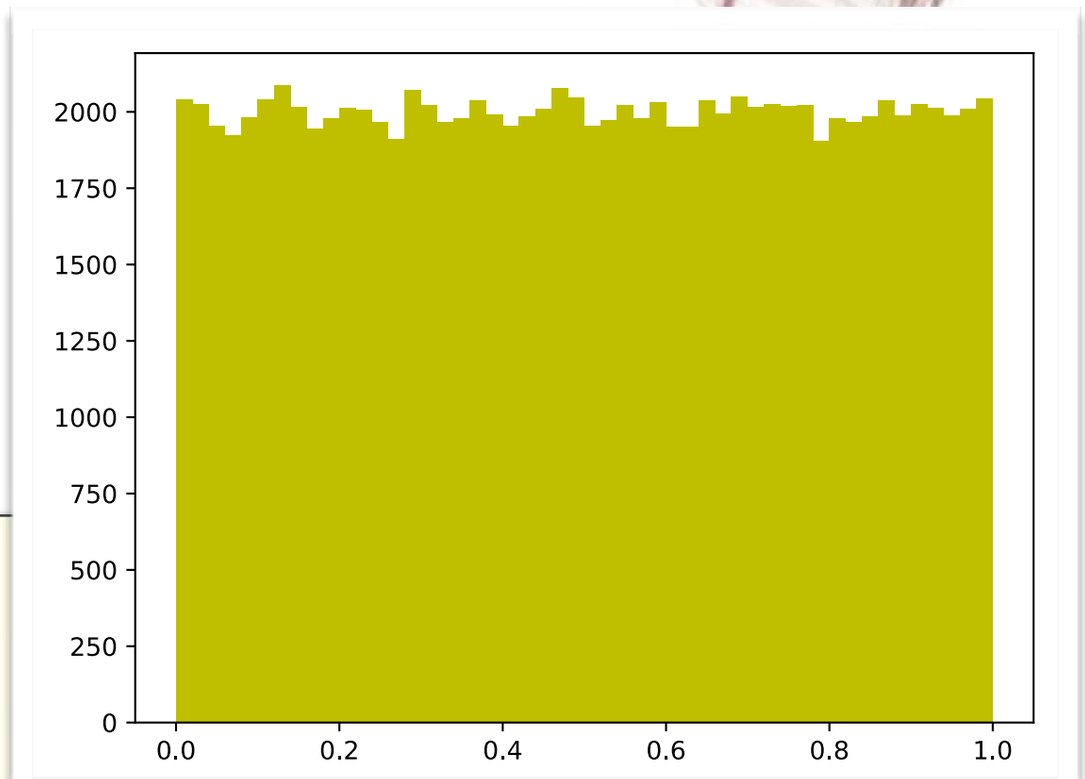
- The LCG generators should produce **an uniform distribution** of random numbers in **[0, M–1]** (or [1, M–1]) Making a plot is the most straightforward way to verify this.
- Let's add a new method in our class that can "mimic" the regular random() function in the random module, which should return a float point number between 0 and 1.

```
class rnd:
    def __init__(self, s = 1234):
        self.seed = s

    def gen(self):
        self.seed = (16807*self.seed) % (2**31-1)
        return self.seed

    def random(self): ⇓ Just normalized by the value of M
        return float(self.gen())/(2**31-1)
```

l207-example-01b.py (partial)

PLOTTING THE DISTRIBUTION (II)

- Simply do some statistical accounting with histograms — which should be **uniformly** distributed between 0 and 1.

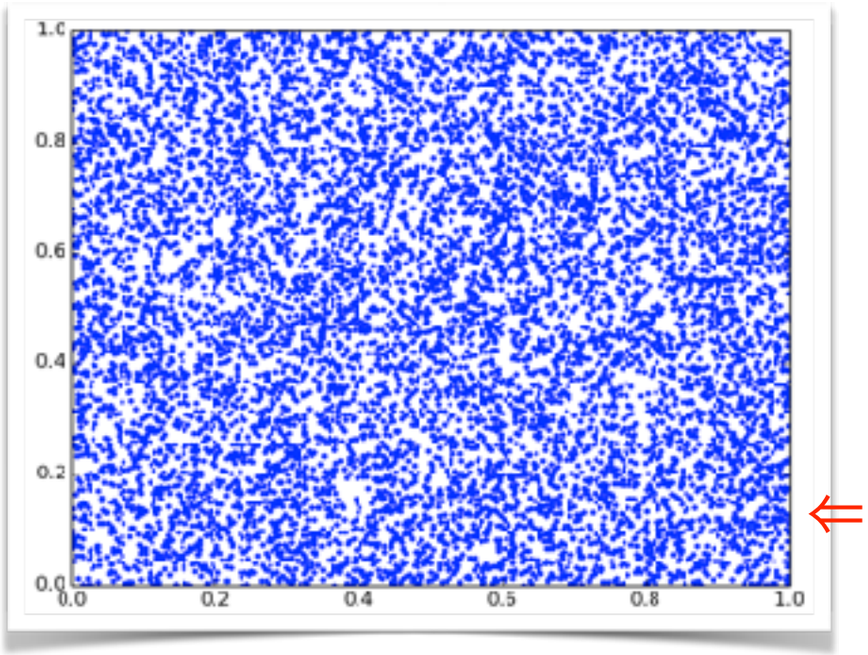



```
import numpy as np
import matplotlib.pyplot as plt

r = rnd()

data = np.zeros(100000)
for i in range(len(data)):
    data[i] = r.random()

plt.hist(data, bins=50, range=(0.,1.), color='y')
plt.show()
```

l207-example-01b.py (partial)

INTERMISSION

- You may try some what different parameters in the LCG generators, ie., different **a**, **c**, and **M**. See if you are able to find a good set of parameters (by chance)?
- As a simple practice — making a 2D scatter plot and set both x an y are uniformly distributed random numbers. See if they are really uniformly distributed in 2D plane?



⇐ Something like this!

BITWISE OPERATIONS

- **OR**:

```
    0101(=5)
OR  0011(=3)
=   0111(=7)
```

```
>>> n = 5 | 3
>>> n
7
```

- **AND**:

```
     0101   (=5)
AND  0011   (=3)
=    0001   (=1)
```

```
>>> n = 5 & 3
>>> n
1
```

- **XOR**:

```
     0101   (=5)
XOR  0011   (=3)
=    0110   (=6)
```

```
>>> n = 5 ^ 3
>>> n
6
```

- **LEFT-SHIFT**:

```
   010111  (=23) << 1
=  101110  (=46)
```

```
>>> 23 << 1
46
```

- **RIGHT-SHIFT**:

```
   010111  (=23) >> 1
=  001011  (=11)
```

```
>>> 23 >> 1
11
```

Remark: bitwise operators are different from logic operators!

HEXADECIMAL/BINARY REPRESENTATION

- Any number starting from **"0x"** (zero-x) is represented in hexadecimal. For example:

```
0x01        = 1
0x05        = 5
0x0a        = 10
0x0b        = 11
0x0f        = 15
0x1f        = 31
0xff        = 255
0xffff      = 2^16-1 = 65535
0xffffffff  = 2^32-1 = 4294967295
```

Sometimes it is very useful to use hexadecimal representations, especially to align a number along the bitwise operations.

HEXADECIMAL/BINARY REPRESENTATION (II)

- Similarly any numbers starting from **"0b"** (zero-b) are in binary format. For example:

```
0b01                = 1
0b00001             = 1
0b10                = 2
0b11                = 3
0b1010              = 10
0b1111              = 15
0b11111111          = 255
0b100000000         = 1<<8 = 256
0b1000000000000     = 1<<12 = 4096
```

In python those representations can be converted by the built-in functions **hex()** or **bin()**.

GENERATOR WITH BITWISE OPERATION:
XORSHIFT

- The algorithm is also simple: **XOR + SHIFT** operation × 3 times:

$$x \leftarrow x \ xor \ (x \gg a_1)$$
$$x \leftarrow x \ xor \ (x \ll a_2)$$
$$x \leftarrow x \ xor \ (x \gg a_3)$$

a_1, a_2, a_3 : bit shifts
Period could be of **$O(2^{32,\,64} - 1)$** if we select good constants!
The only missing number is zero...

- Some good selection of **a_1**, **a_2**, and **a_3** :

a_1	a_2	a_3
21	35	4
20	41	5
17	31	8
11	29	14

. . .

For example:

$$\begin{aligned} \text{initial}: \quad & x = 1234 \\ x \leftarrow x \ xor \ (x >> 21): \quad & x = 1234 \\ x \leftarrow x \ xor \ (x << 35): \quad & x = 42399917147346 \\ x \leftarrow x \ xor \ (x >> 4): \quad & x = 40651865457823 \end{aligned}$$

JUST TRY IT OUT

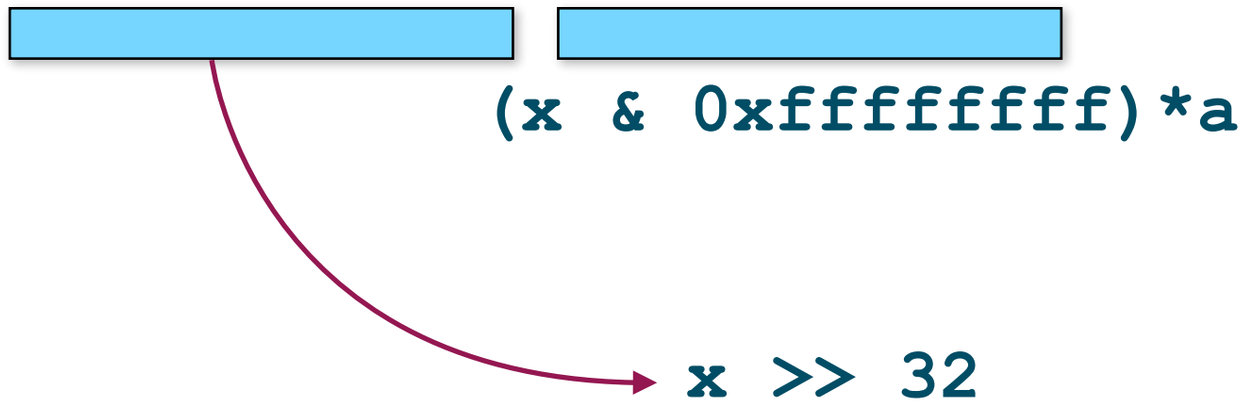
```python
class rnd:
    def __init__(self, s = 1234):
        self.seed = s

    def gen(self):
        self.seed = self.seed ^ (self.seed >> 21)
        self.seed = self.seed ^ (self.seed << 35)
        self.seed = self.seed ^ (self.seed >>  4)
        return (self.seed & 0xffffffff)⇐ output lower 32 bits

r = rnd()
for i in range(30):
    print(r.gen(), end=' ')
```

l207-example-02.py

```
First few numbers (w/ seed = 1234)
1183, 288731222, 1003807570, 2737978148, 560314590, 860181832,
2874940191, 481057057, 4263710680, 1133242283, 547512940, ...
```

MULTIPLY WITH CARRY – MWC METHOD

- For updating a 64-bit unsigned integer:



take lower 32 bits
and scale it by a constant **a**

shift higher 32 bits
to lower part

- Some good selection of **a**:

a
4294957665
4294963023
4162943475
3947008974

• • •

Add them (high, low 32 bits) together to get the next iteration!

Period could be of **O[(2^{32} a – 2)/2]**

A QUICK IMPLEMENTATION

```
class rnd:
    def __init__(self, s = 1234):
        self.seed = s

    def gen(self):
        self.seed = \
(self.seed & 0xffffffff)*4294957665 + (self.seed>>32)
        return (self.seed & 0xffffffff)⇐ output lower 32 bits as well!

r = rnd()
for i in range(30):
    print(r.gen(), end=' ')
```

l207-example-03.py

```
First few numbers (w/ seed = 1234)
4283082642, 2791954211, 1467339856, 1284198655, 2855902741,
1055460788, 3900636741, 2101943962, 2259196020, 2089392165, ...
```

AN EASY WAY TO IMPROVE YOUR GENERATOR

- Those algorithms we have introduced have a limit of **period < 2^{64} (~1.8×10^{19})**.
- It will not be too good if you want to produce really huge amount random of numbers.
- The idea is simple: **combine different generators**.

Stronger if you combine?

COMBINATION: XORSHIFT + MWC

We support to have a period of **O(9 x 10^{37})**...

```
class rnd:
    def __init__(self, s1 = 1234, s2 = 5678):
        self.s1 = s1
        self.s2 = s2

    def gen(self):
        self.s1 = self.s1 ^ (self.s1 >> 17)
        self.s1 = self.s1 ^ (self.s1 << 31)
        self.s1 = self.s1 ^ (self.s1 >>  8)
        self.s2 = (self.s2 & 0xffffffff)*4294957665 +
                  (self.s2>>32)
        return ((self.s1 ^ self.s2) & 0xffffffff)
```

Combining generators with **XOR** ⇑

l207-example-04.py (partial)

```
First few numbers (w/ seed = 1234, 5678)
2512230328, 3081706301, 1968151581, 3266289562, 2076928125,
1738967221, 2657722299, 1153926827, 738420122, 2921085993, ...
```

IF YOU ARE NOT SATISFIED WITH 10^{38}...

- A generator with a very large period – **Mersenne Twister**:
  - It was developed in 1997 by Makoto Matsumoto (松本真) and Takuji Nishimura (西村拓士). The webpage (you can download the source code, which is written in C):
    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 famous one. Most important — it's already included in the python random module as well as NumPy!

You can simply use the generator coming from python libraries or NumPy.

INTERMISSION

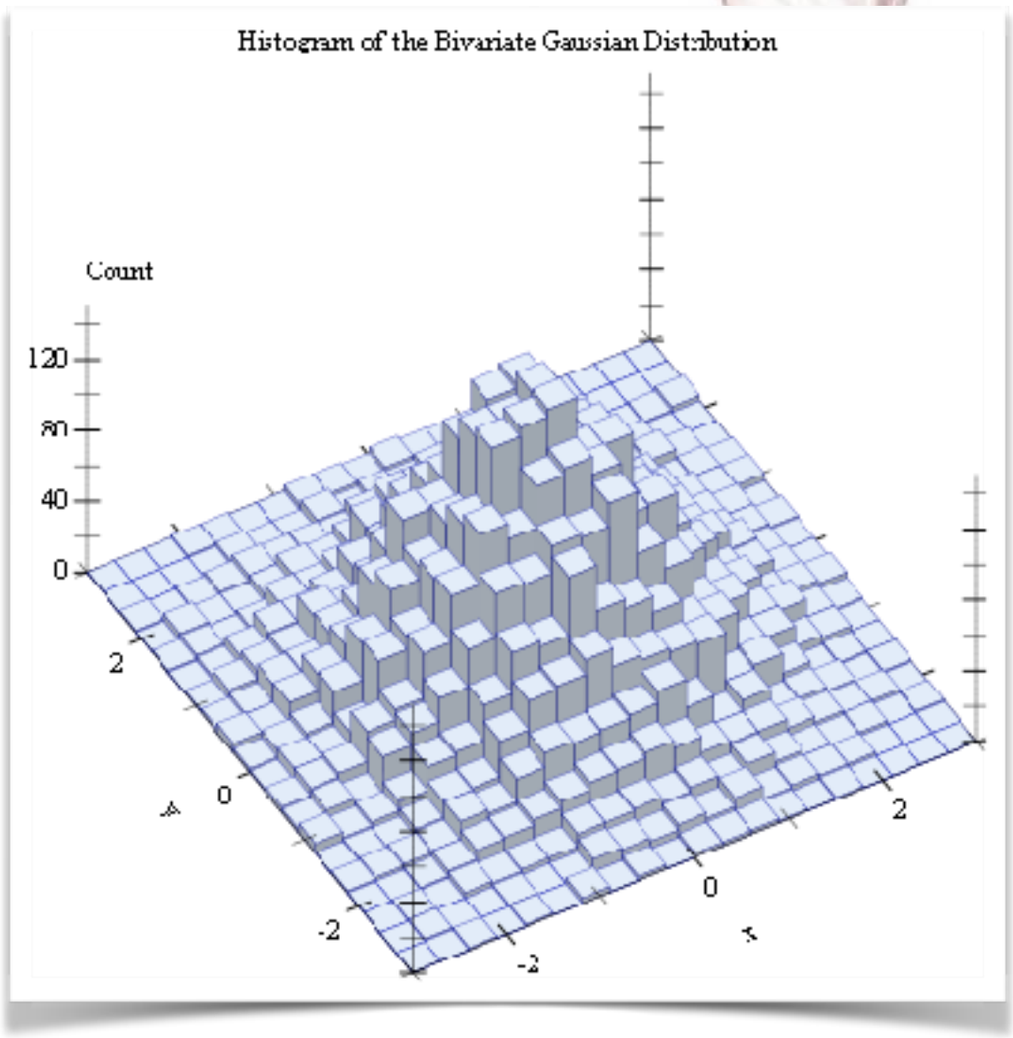
- If you are not so familiar with the bit-wise operations as well as hexadecimal representation, it is a good timing to practice a little bit more. For example:

```
37 (0b0100101) | 73 (0b1001001) = ?
37 (0b0100101) & 73 (0b1001001) = ?
37 (0b0100101) ^ 73 (0b1001001) = ?
```

- You can always use bulit-in bin() function to verify the calculations above!

GENERATING NONUNIFORM DISTRIBUTIONS

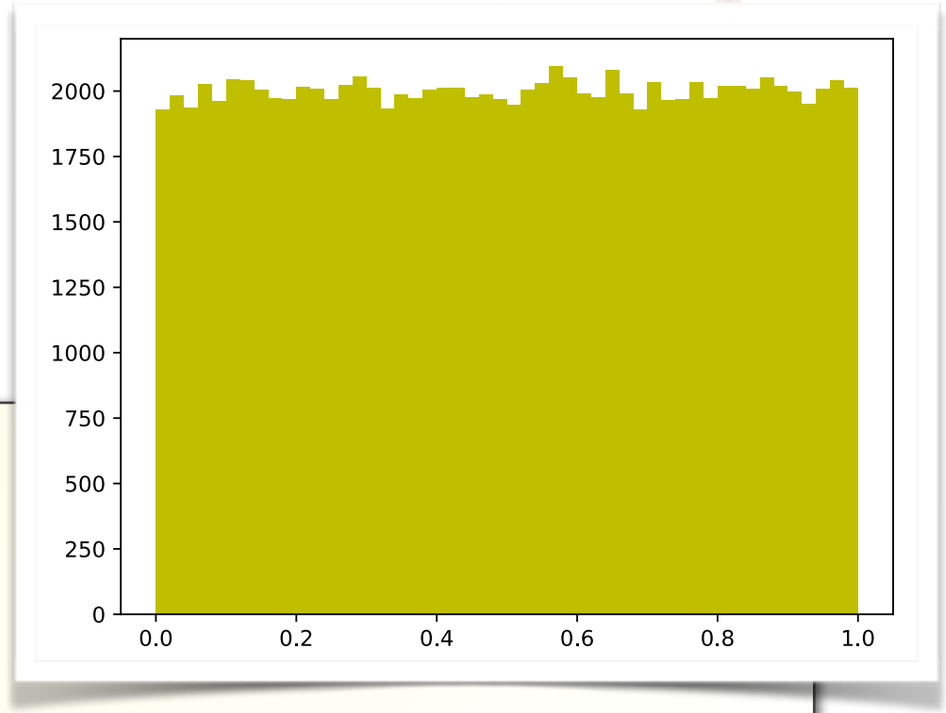
- In many practical cases we need to generate the random numbers according to a desired distribution. You have already well experienced these functionalities?



Let's discuss how to convert those flat random numbers to a different shape!

THE STARTING POINT:
UNIFORM RANDOM DISTRIBUTION

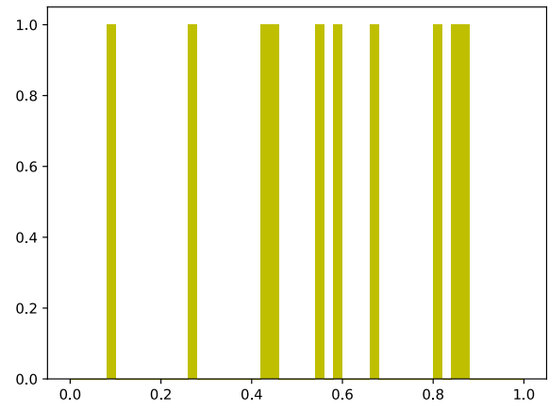
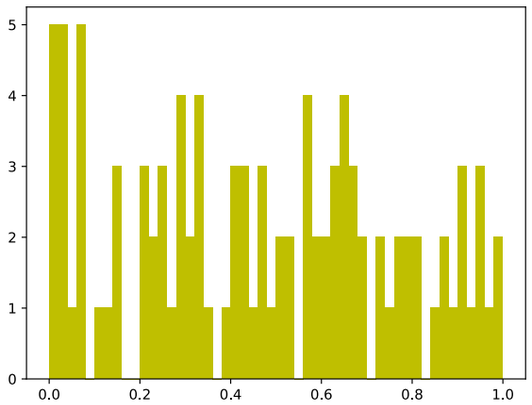
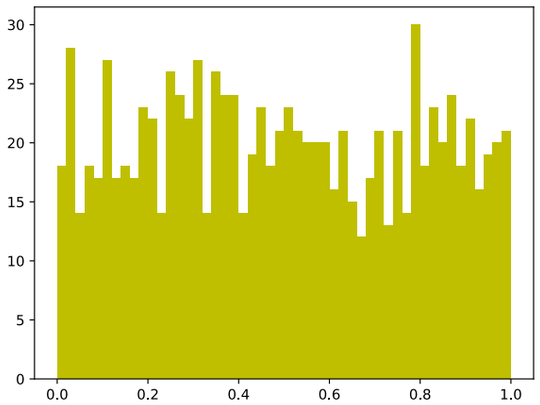
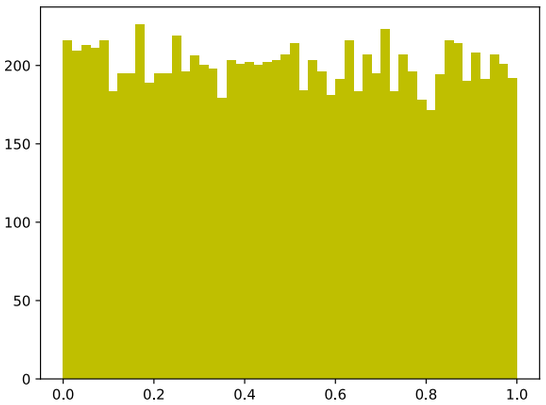
- The small piece of code below will produce an uniform random distribution as we did in l12-example-01b.py, but based on the random number generator from NumPy.
- This is the starting point of all the discussions below!



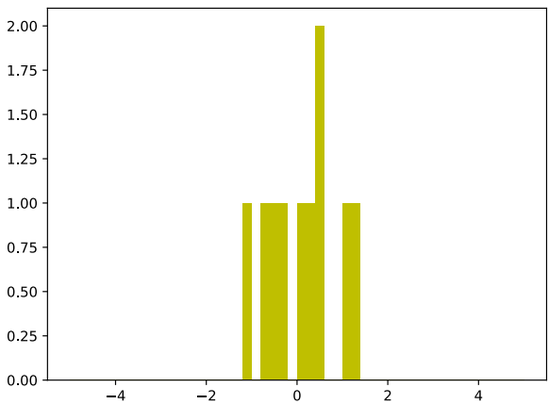
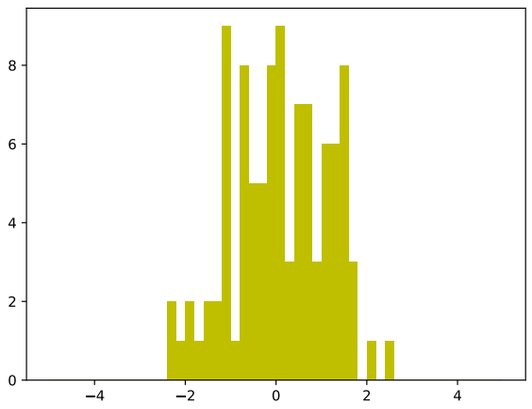
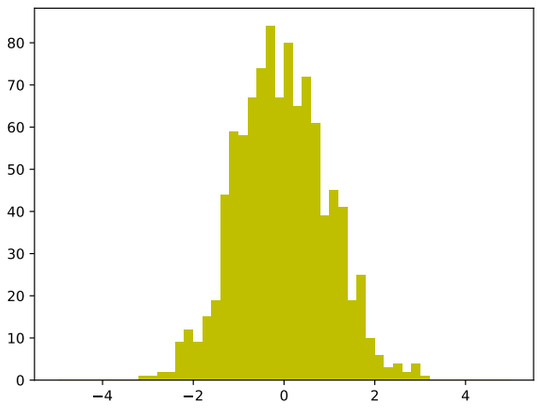
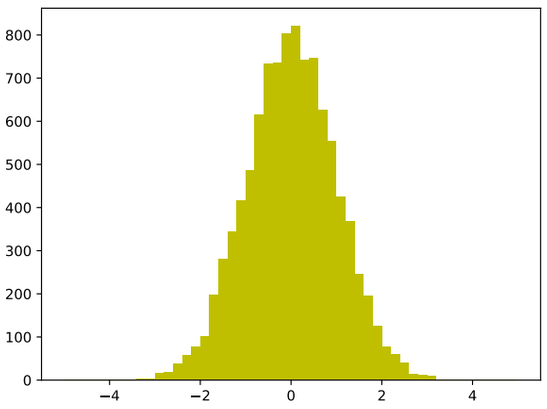

```
import numpy as np
import matplotlib.pyplot as plt

data = np.random.rand(100000)

plt.hist(data, bins=50, range=(0.,1.), color='y')
plt.show()
```

ACCUMULATE RANDOM DISTRIBUTIONS

- Uniformly random distribution: **`random.rand()`**:

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

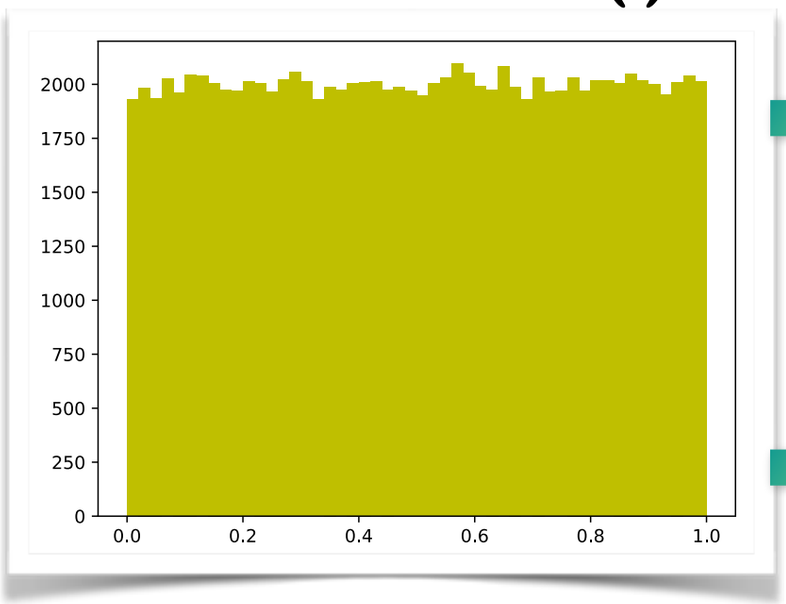





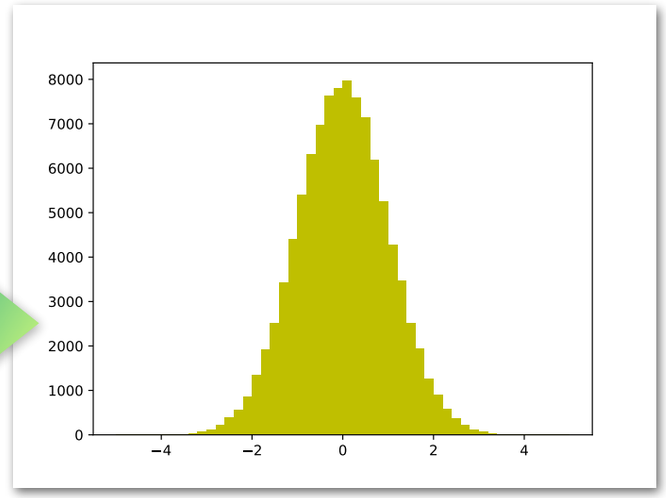
GENERATION OF NON-UNIFORM DISTRIBUTIONS

- The basic idea is to produce any non-uniform distributions based on the uniform one, as far as **the function form is given/can be calculated**!

Uniform source

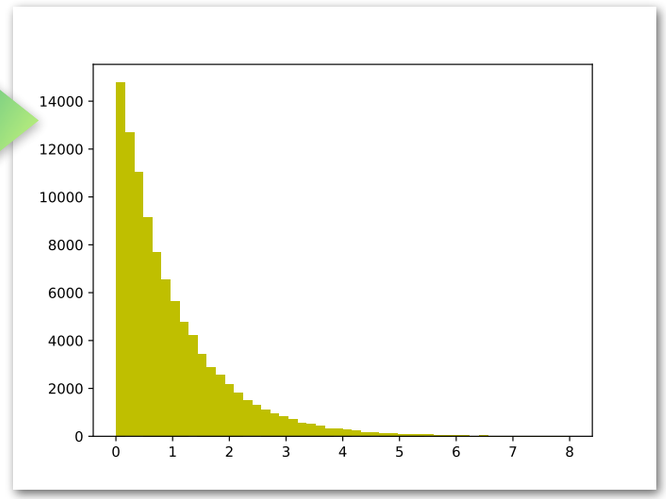
`random.rand()`





Gaussian

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



Exponential

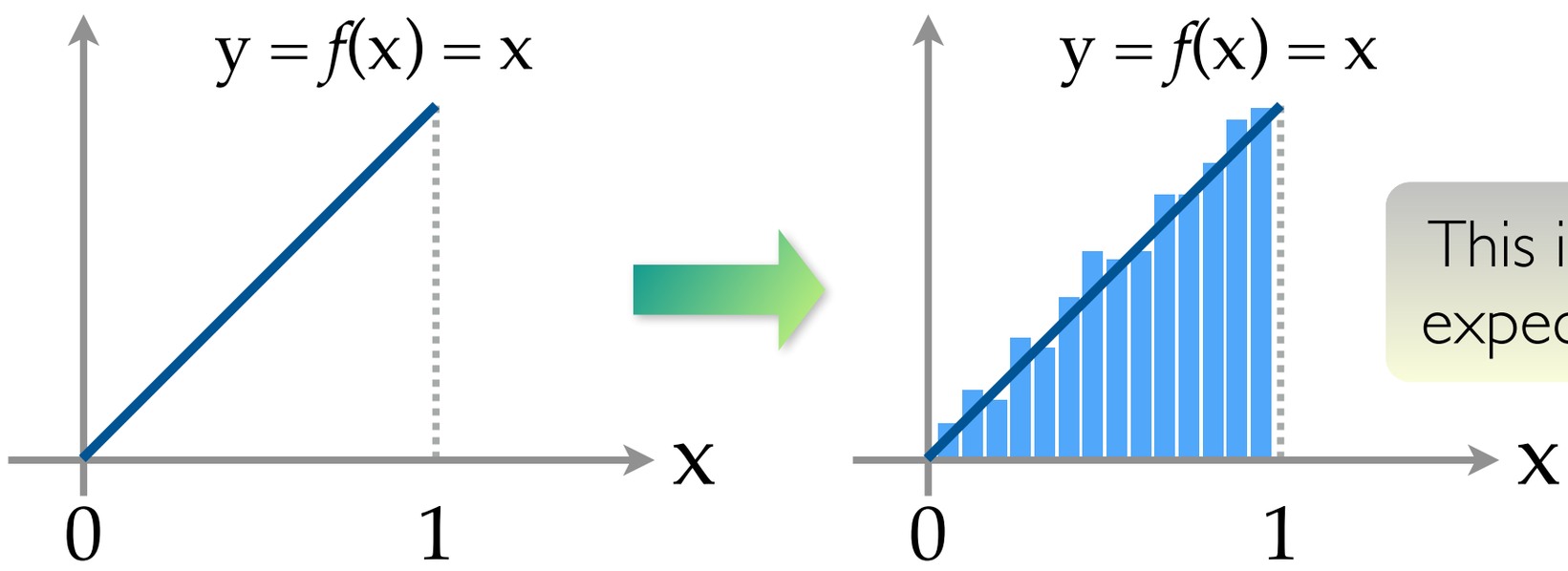
$$f(x) = \exp\left(-\frac{x}{\tau}\right)$$

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:

$$f(x) = x$$

- The first step is to draw the target function and observe its shape.



GENERATION OF NON-UNIFORM DISTRIBUTIONS (III)

- Based on an input of uniformly distributed random numbers, now we want to convert the distribution to follow the given function.
- Probably 90% of the people will start with such a naive code by just inserting the random variables into the function?

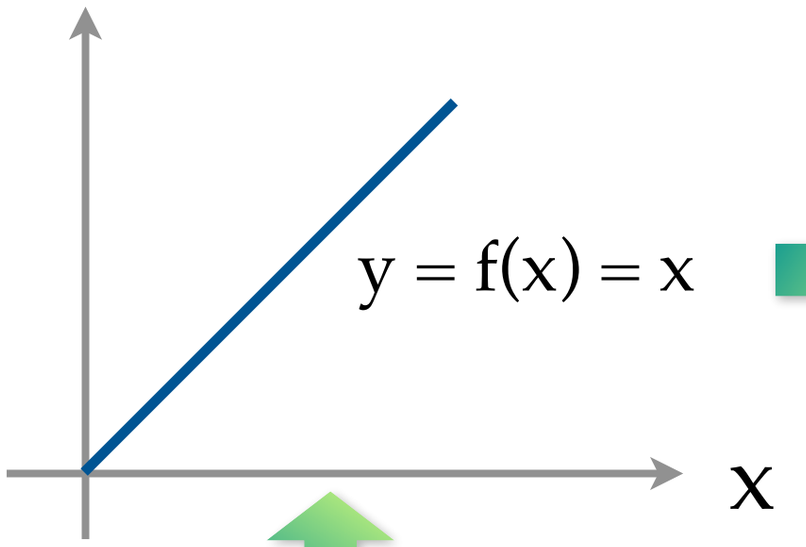
```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x

x = np.random.rand(100000)
data = f(x) ⇐ Simply inject x into f(x)?

plt.hist(data, bins=50, range=(0.,1.), color='y')
plt.show()
```

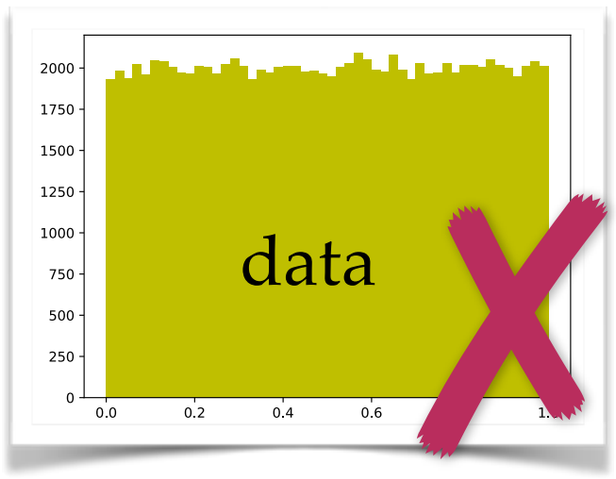
This is definitely incorrect...

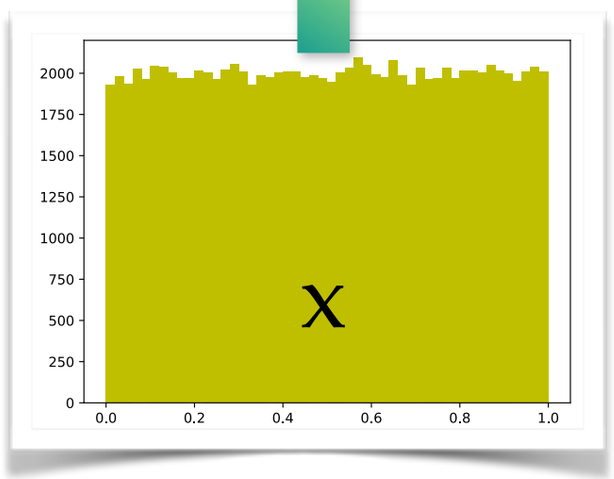
l207-example-05.py (partial)

GENERATION OF NON-UNIFORM DISTRIBUTIONS

- 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.

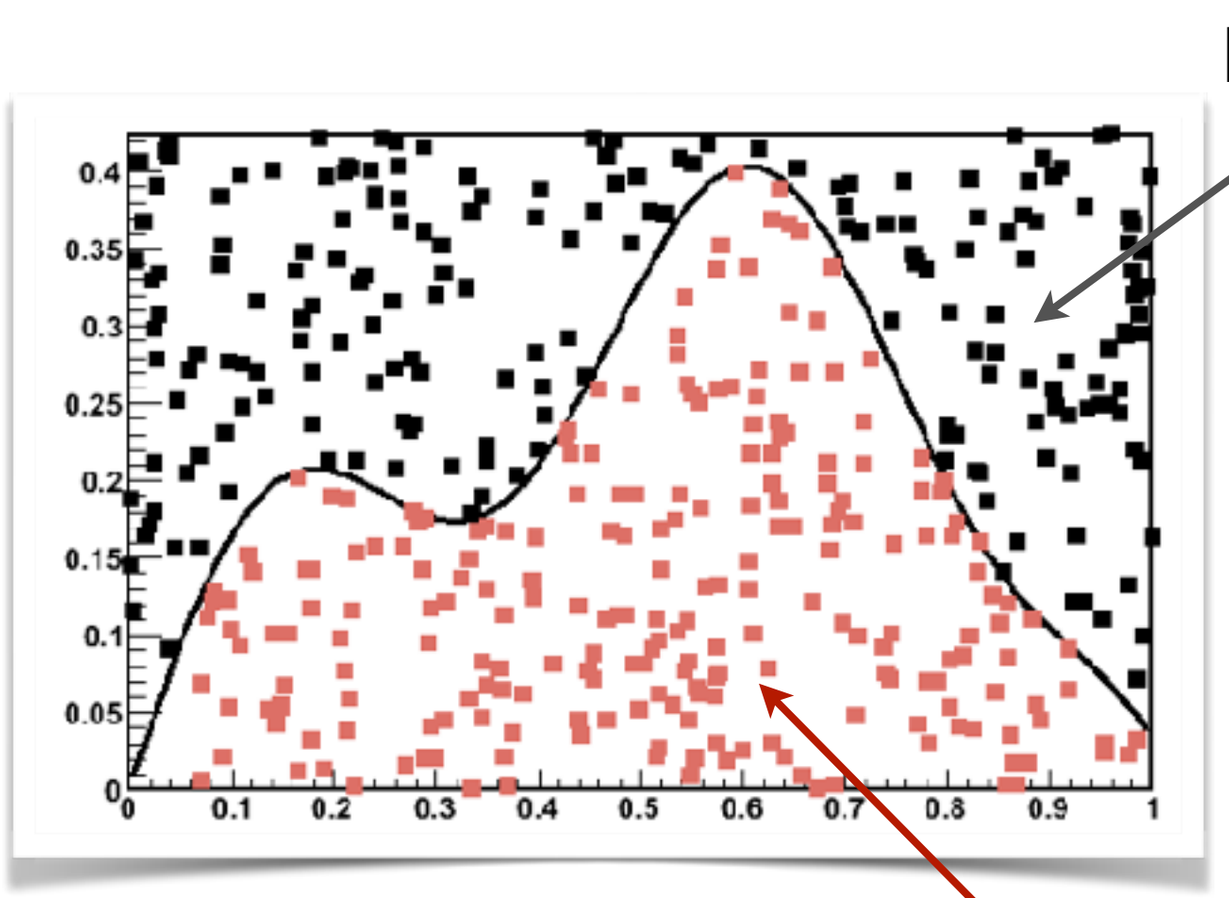
SO WHAT SHOULD WE DO?

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

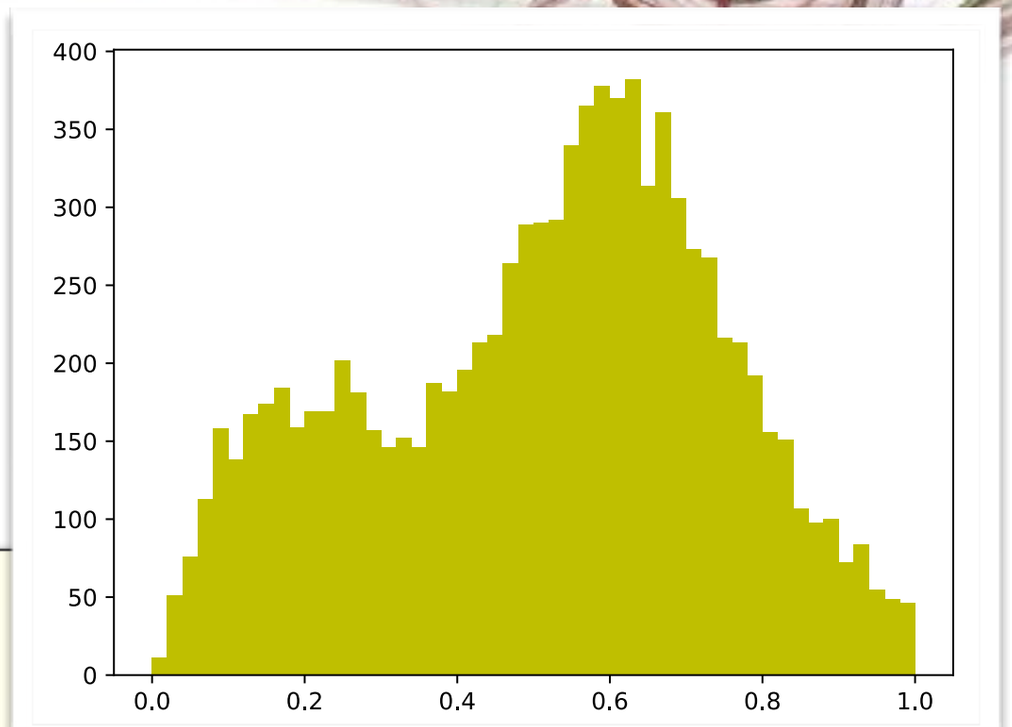
- This is one of the most simplest algorithms, it's quite inefficient, but still very useful! The trick is simply
  Instead of 1D — generate the random numbers **uniformly in 2D**:

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



A QUICK IMPLEMENTATION

- All we need to do is generating x and y uniformly and only keep the values of x if **y<f(x)**.




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

def f(x):
    return x - x**2 + x**3 - x**4 + np.sin(x*13.)/13.

x = np.random.rand(20000)
y = np.random.rand(20000)*0.45
data = x[y<f(x)]

plt.hist(data, bins=50, range=(0.,1.), color='y')
plt.show()
```

⇐ This is since we know the maximum value of the function is below 0.45

l207-example-06.py

A QUICK IMPLEMENTATION (II)

- If we have no knowledge of the function maximum, what could we do?
  - ➡ Just use random number to scan/guess the maximum value, i.e. **"importance sampling"**.

Remember we still assume we have no knowledge about the exact function form!

```
def f(x):
    return x - x**2 + x**3 - x**4 + np.sin(x*13.)/13.

f_max = f(np.random.rand(1000)).max()*1.1

x = np.random.rand(20000)
y = np.random.rand(20000)*f_max
data = x[y<f(x)]

plt.hist(data, bins=50, range=(0.,1.), color='y')
plt.show()
```

⇑ Using 1000 trials to get the maximum value + 10% protection!

I207-example-06a.py (partial)

YOU CAN ALSO DO THIS

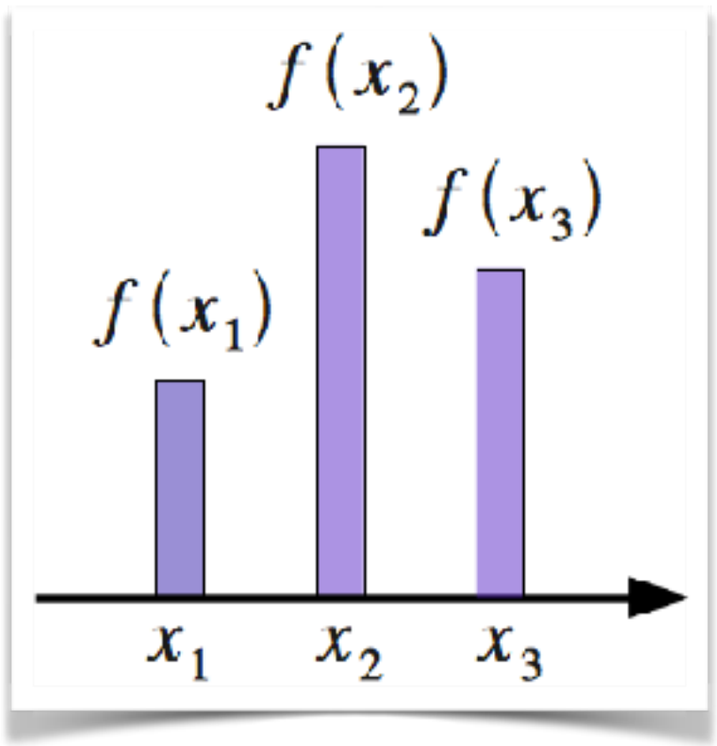
- The value given by the function f(x) is actually the the **"weight"** of each event, i.e. *the probability that the event should be accepted or not*.
- One can apply an **"unweighting"** procedure to get the desired random distribution.
- This is in particular useful for generating random numbers with a very complicated multi-dimensional function.

```
def f(x):
    return x - x**2 + x**3 - x**4 + np.sin(x*13.)/13.

x = np.random.rand(20000)  ⇐ The pair of x,w are called "weighted events".
w = f(x)

y = np.random.rand(20000)  ⇐ unweighting: accept x or not according to y
data = x[y<w/w.max()]
```

l207-example-06b.py (partial)

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:



Generate r∈[0,1] uniformly:

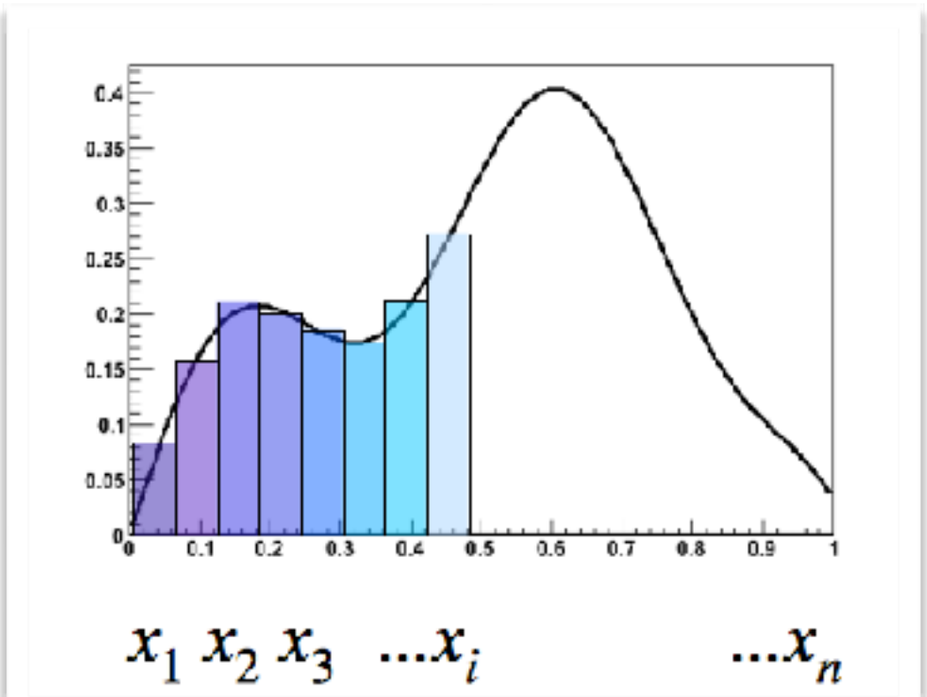
$$\text{if } r < \frac{f(x_1)}{f(x_1)+f(x_2)+f(x_3)} \text{ then } x = x_1 \text{ else}$$

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

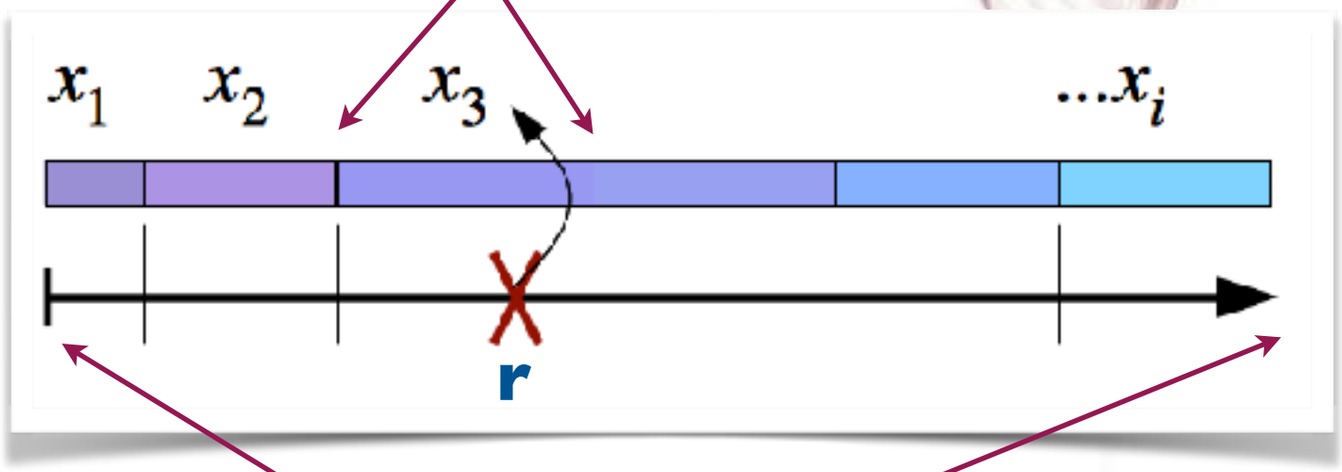
$$\text{if } r < \frac{f(x_1)+f(x_2)+f(x_3)}{f(x_1)+f(x_2)+f(x_3)} \text{ then } x = x_3$$

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

WITH EXPLICIT MATHEMATICS

- Take the continuous limit:

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

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

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

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

$$W(x) = \frac{\int_a^{x} f(x')dx'}{\int_a^b f(x')dx'} = r$$

Find the invert function of W(x)

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

A STRAIGHTFORWARD EXAMPLE

For example:

$$f(x) = \exp(-x);\ [a,b] = [0,1] \Rightarrow \int_0^x \exp(-x')dx' = 1 - \exp(-x)$$

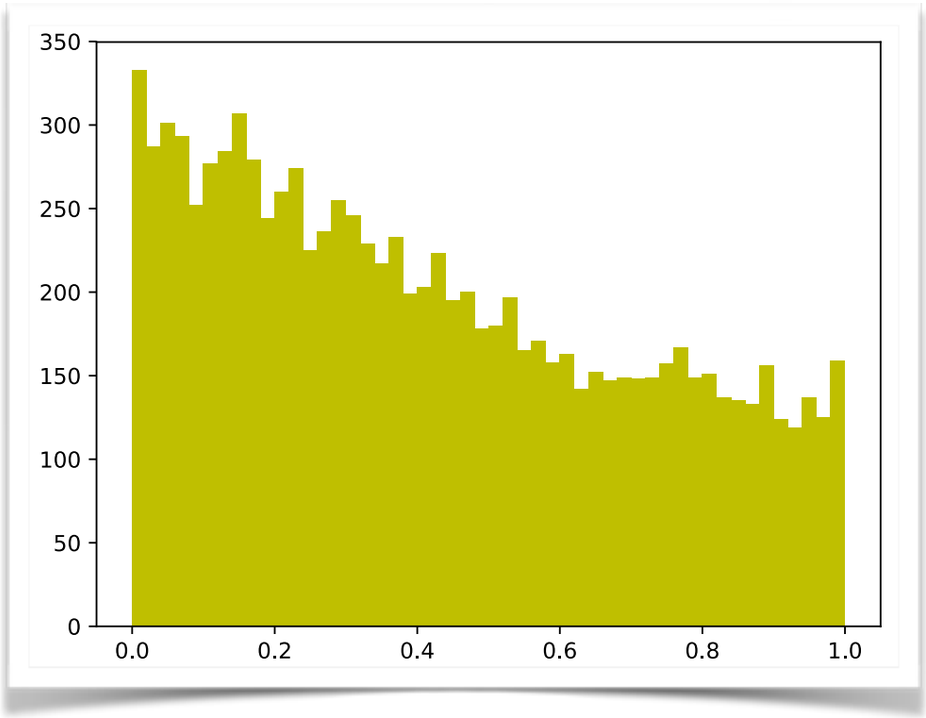
$$W(x) = \frac{\int_0^x \exp(-x')dx'}{\int_0^1 \exp(-x')dx'} = \frac{1 - \exp(-x)}{1 - \exp(-1)}$$

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

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

```
r = np.random.rand(10000)
data = -np.log(1.-r+r/np.exp(1.))

plt.hist(...)
```

l207-example-07.py (partial)



EXTENDED TO INFINITY?

- This method also works with infinite bounds! e.g.

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

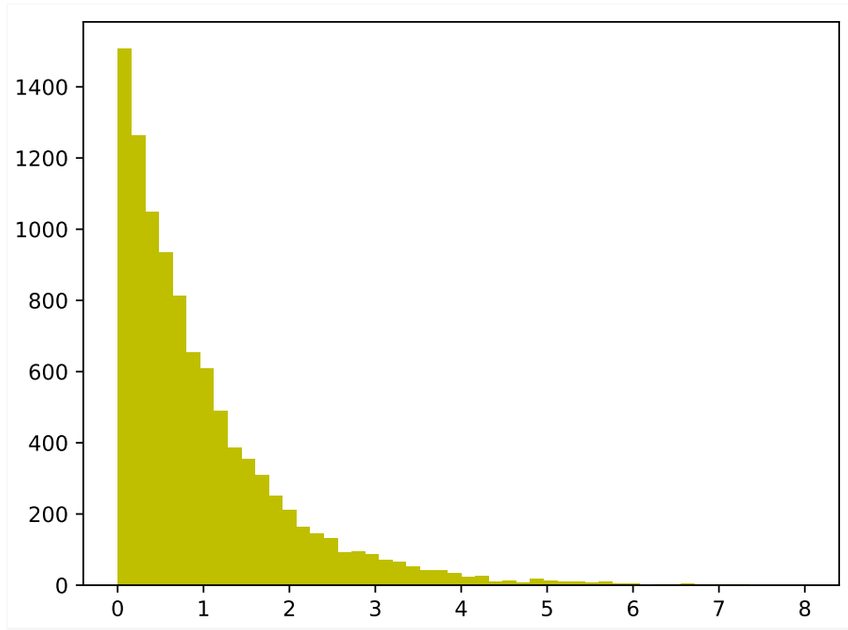
$$W(x) = \frac{\int_0^x \exp(-x')dx'}{\int_0^\infty \exp(-x')dx'} = 1 - \exp(-x)$$

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

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

```
r = np.random.rand(10000)
data = -np.log(1.-r)

plt.hist(...)
```

l207-example-07a.py (partial)



GENERATING A GAUSSIAN

- Instead of the simple hit-or-mass method, let's practice the inverse transformation with the **Gaussian** 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

GENERATING A GAUSSIAN (II)

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

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

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

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

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

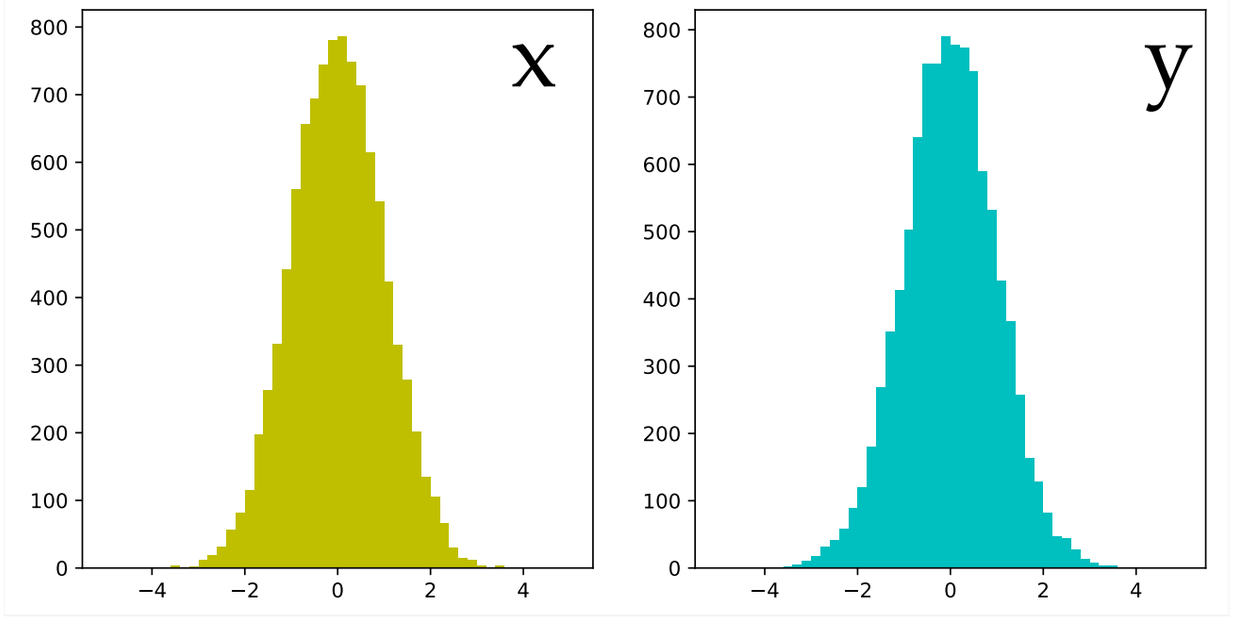
or

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

Since it does not matter if we generate r_1 *or* $1-r_1$

GENERATING A GAUSSIAN (III)

- See how straightforward the generation is!
- Both x & y can be used (no waste of random numbers)

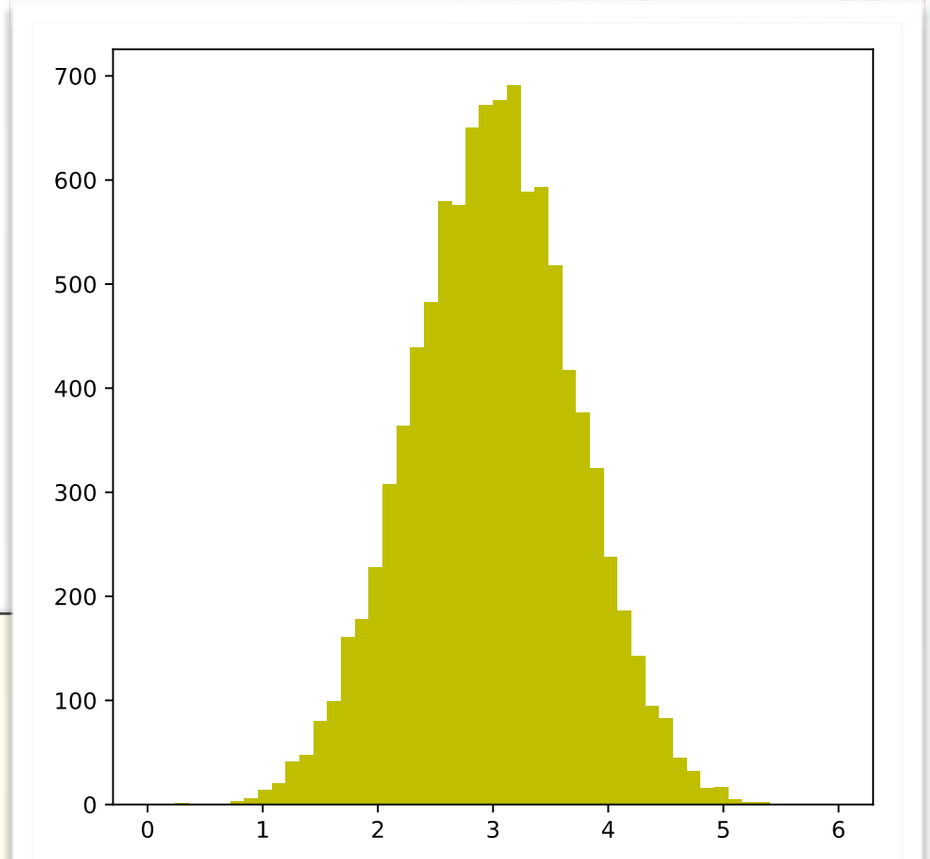



```
r1 = np.random.rand(10000)
r2 = np.random.rand(10000)
x = (-2.*np.log(r1))**0.5*np.cos(2.*np.pi*r2)
y = (-2.*np.log(r1))**0.5*np.sin(2.*np.pi*r2)

plt.subplot(1,2,1)
plt.hist(x, bins=50, range=(-5.,+5.), color='y')

plt.subplot(1,2,2)
plt.hist(y, bins=50, range=(-5.,+5.), color='c')

plt.show()
```

l207-example-08.py (partial)

ALTERNATIVELY…

- By simply adding multiple uniform random numbers you can also approximate a Gaussian distribution.
- This is just a demonstration of **central limit theorem** in fact!




```
r = np.random.rand(10000)
r += np.random.rand(10000)
r += np.random.rand(10000)
r += np.random.rand(10000)
r += np.random.rand(10000)
r += np.random.rand(10000)

plt.hist(r, bins=50, range=(0.,+6.), color='y')
plt.show()
```

l207-example-08a.py (partial)

COMMENTS

- There are actually many different methods to generate Gaussian distributions. The one we introduced is just one of the "classical" methods widely used in various places.
- Besides Gaussian, there are many generators for special functions implemented in **`numpy.random`** module. It would be useful to go through them once and use them when needed.
- On the other hand, now you should have the capability to produce any random distribution as you wish. This is extremely useful for various studies, such as **Monte Carlo** studies or **statistical tests**.

More to be discussed in the later lecture!

APPLICATION: MONTE CARLO INTEGRATION

- Usually it is very difficult to do a numerical integration over high-dimensions. e.g. by cutting one dimensional space to 1000 steps, for 10 dimensions — it will just take 1000^{10} ($=10^{30}$) operations!
- This is the place for the **Monte Carlo integration** to cut in.
- The math is actually trivial (*integration by mean value*):

$$I = \int_a^b f(x)dx = (b-a)\cdot\langle f\rangle \qquad \rightarrow \ \langle f\rangle \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

$$\sigma^2 \approx \frac{1}{N-1}\sum_{i=1}^{N}(f(x_i)-\langle f\rangle)^2$$

One only needs to use random numbers to calculate the mean value among the desired space.

A DEMO APPLICATION

- Let's take a "simple" function of 6 dimensions:

$$f(a,b,c,d,e,f) = \sin(a) + \sin(2b) + \sin(3c) + \cos(d) + \cos(2e) + \cos(3f)$$

$$I = \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 \int_0^1 f(a,b,c,d,e,f)da\ db\ dc\ dd\ de\ df = ?$$

$$\int f d\Omega =$$

Well, we do know the exact integration of this example function:

$$-\left[\cos(a)\right]_0^1 - \left[\frac{\cos(2b)}{2}\right]_0^1 - \left[\frac{\cos(3c)}{3}\right]_0^1 + \left[\sin(d)\right]_0^1 + \left[\frac{\sin(2e)}{2}\right]_0^1 + \left[\frac{\sin(3f)}{3}\right]_0^1$$

But generally it will be very difficult for more complicated functions (e.g. *real 6 dimensional function with cross terms!*)

A DEMO APPLICATION (II)

- A quick implementation can be prepared easily:

```
def func(a, b, c, d, e, f):  ⇐ the 6D function
    return np.sin(a)+np.sin(b*2.)+np.sin(c*3.)+np.cos(d)
+np.cos(e*2.)+np.cos(f*3.)

def intfunc(a, b, c, d, e, f):  ⇐ the exact integration
    return -np.cos(a)-np.cos(b*2.)/2.-np.cos(c*3.)/3.+np.sin(d)
+np.sin(e*2.)/2.+np.sin(f*3.)/3.

intf_exact = intfunc(1.,1.,1.,1.,1.,1.)-intfunc(0.,0.,0.,0.,0.,0.)
print("Exact  = %+.5f" % intf_exact)

nsamples = 1000000 # 1 million trials
v = np.random.rand(nsamples,6)
val = func(v[:,0],v[:,1],v[:,2],v[:,3],v[:,4],v[:,5])
intf_rand = val.sum() / nsamples
intf_rand_err = (((val**2).sum()/nsamples-intf_rand**2)/(nsamples-1))**0.5

print("Random = %+.6f +- %.6f" % (intf_rand,intf_rand_err))
print(" (diff = %+.6f)" % (intf_rand-intf_exact))
```

$\langle f \rangle$ & σ

l207-example-09.py (partial)

A DEMO APPLICATION (III)

```
sep = np.linspace(0.025,0.975,20)
va,vb,vc,vd,ve,vf = np.meshgrid(sep,sep,sep,sep,sep,sep)
intf_boxes = func(va,vb,vc,vd,ve,vf).sum() * 0.05**6

print("Boxes  = %+.6f" % intf_boxes)
print(" (diff = %+.6f)" % (intf_boxes-intf_exact))
```

l207-example-09.py (partial)

As a comparison, let's also do a simple "boxes" numerical integration. If we take 20 slices per dimension, the total required number of function call is $20^6 = 64{,}000{,}000$ times.

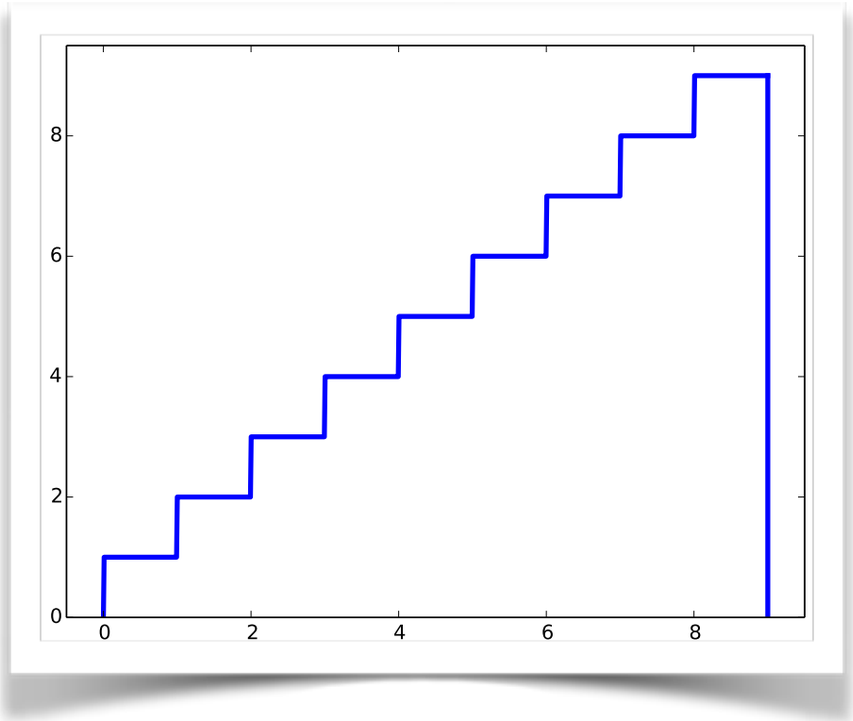
```
Exact  = +3.17426
Random = +3.175567 +- 0.000964
 (diff = +0.001305)
Boxes  = +3.175548
 (diff = +0.001287)
```

MC integration is actually easier and quicker in the case of high-dimensional integration!

HANDS-ON SESSION

- **Practice 01:**
  Generate the random distribution of >10k events for the following step function for x in the range **[0,9]**:

$$f(x) = 1 \text{ if } 0 < x \leq 1$$
$$= 2 \text{ if } 1 < x \leq 2$$
$$\dots$$
$$= 9 \text{ if } 8 < x \leq 9$$



HANDS-ON SESSION

- **Practice 02:**
  Generate the random distribution of >10k events for the following model of a second order polynomial plus a Gaussian function, which was used in our fitting example:

$$f(x) = \frac{N}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma}\right] + ax^2 + bx + c$$

Where x is in the range of **[101, 182]**.
The other parameters are
$N = 19$, $\mu = 126$, $\sigma = 2$,
$a = -0.0002$, $b = 0.05$, $c = -1.5$