# 993SM - Laboratory of Computational Physics II unit - (a) September 26, 2025

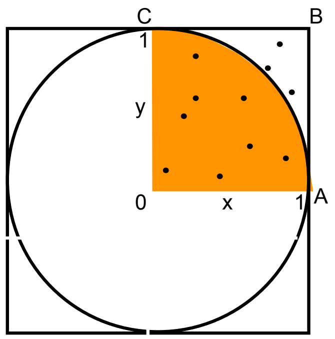
**Maria Peressi**
Università degli Studi di Trieste - Dipartimento di Fisica
Sede di Miramare (Strada Costiera 11, Trieste)
e-mail: peressi@units.it
tel.: +39 040 2240242

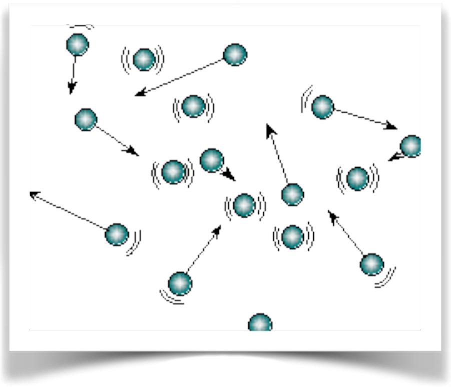
# Random numbers and Monte Carlo(*) Techniques

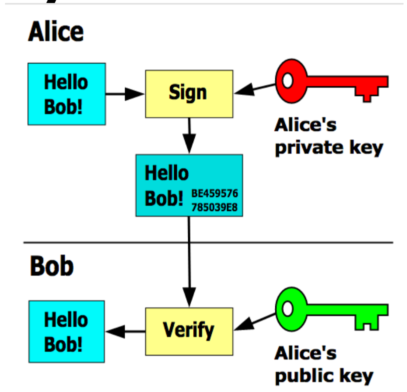
(*) any  procedure making use of random numbers

# Random numbers: use

- in numerical analysis (to calculate integrals)
- to simulate and model complex or intrinsically random phenomena
- to generate data encryption keys
- ...







## (a) Random numbers **with uniform distribution**

Characteristics and Generation

# Pseudo random numbers generation with a computer

"pseudo" because they are necessarily generated with deterministic procedures
(the computer is a deterministic system!)

A sequence of computer generated random numbers is not truly random, since each number is completely determined from the previous one.

But it may "appear" to be random.

# (pseudo)Random numbers generation

These are sequences of numbers generated by computer algorithms, usally in a uniform distribution in the range [0,1].

To be precise, the alogrithms generate integers $I_n$ between 0 and M, and return a real value.

$$x_n = I_n / M$$

the sequence may "appear" to be random

[ Attention: in a Fortran code, write: $x_n$ =float($I_n$)/M !!!]

## (pseudo)random numbers generation: “Linear congruential method (LCM)”

(Lehemer, 1948)

$$\boxed{I_{n+1} = (a\, I_n + c) \bmod m}$$

Starting value (seed) = $I_0$

a, c, and m are specially chosen

$a, c \geq 0$ and $m > I_0, a, c$

“A mod m” is the **remainder** of the division of A by m

# (pseudo)random numbers generation:

## "Linear congruential method (LCM)"

$$\boxed{I_{n+1} = (a\, I_n + c) \bmod m}$$

QUESTIONS:

- in which interval are the pseudorandom numbers generated?
- Can we obtain all the numbers in such interval?
- Is the sequence periodic?
- Which is the period?
- Which is the maximum period?

<u>Limits of the algorithm:</u>

- the "quality" of the sequence is very sensitive to the choice of the parameters
- even if $c \neq 0$

## ☆ Choice of modulus, m

m should be as large as possible since the period can never be longer than m.

One usually chooses m to be near the largest integer that can be represented. On a 32 bit machine, that is $2^{31} \approx 2\times10^9$.

$$\boxed{I_{n+1} = (a\, I_n + c) \bmod m}$$

## ☆ Choice of multiplier, a

It was proven by M. Greenberger in 1961 that the sequence will have period m, if and only if:

i) c is relatively prime to m;
ii) a-1 is a multiple of p, for every prime p dividing m;
iii) a-1 is a multiple of 4, if m is a multiple of 4

# More subtle limits, even of some smart algorithms...

A popular random number generator was distributed by IBM in the 1960's with the algorithm: https://en.wikipedia.org/wiki/RANDU

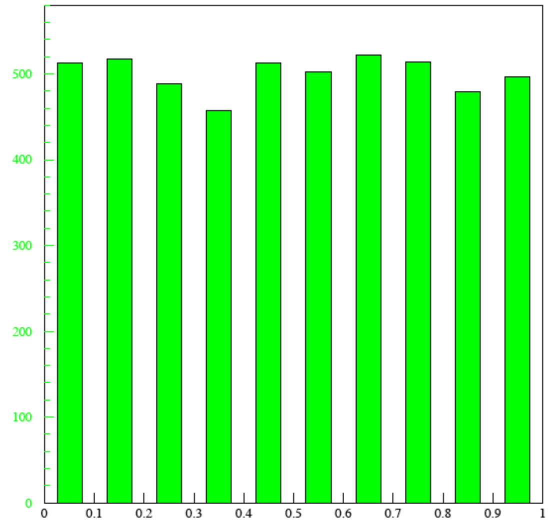
$$I_{n+1} = (65539 \times I_n) \bmod 2^{31}$$

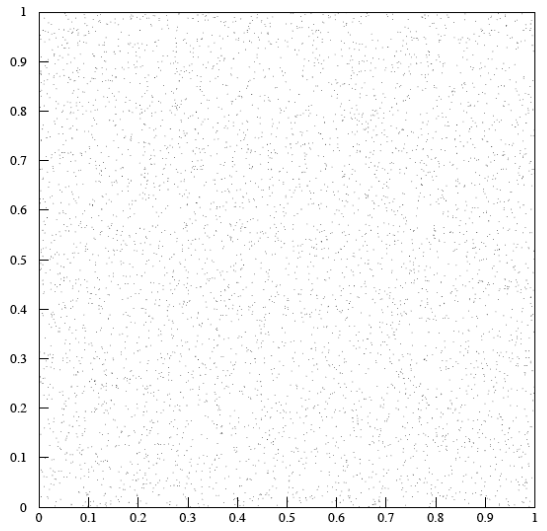
$65539 = 2^{16}+3$; initial seed $I_0$: odd number

1D distribution Looks okay $\longrightarrow$

$\mathbf{x_i, p(x_i)}$

Results from Randu: 1D distribution





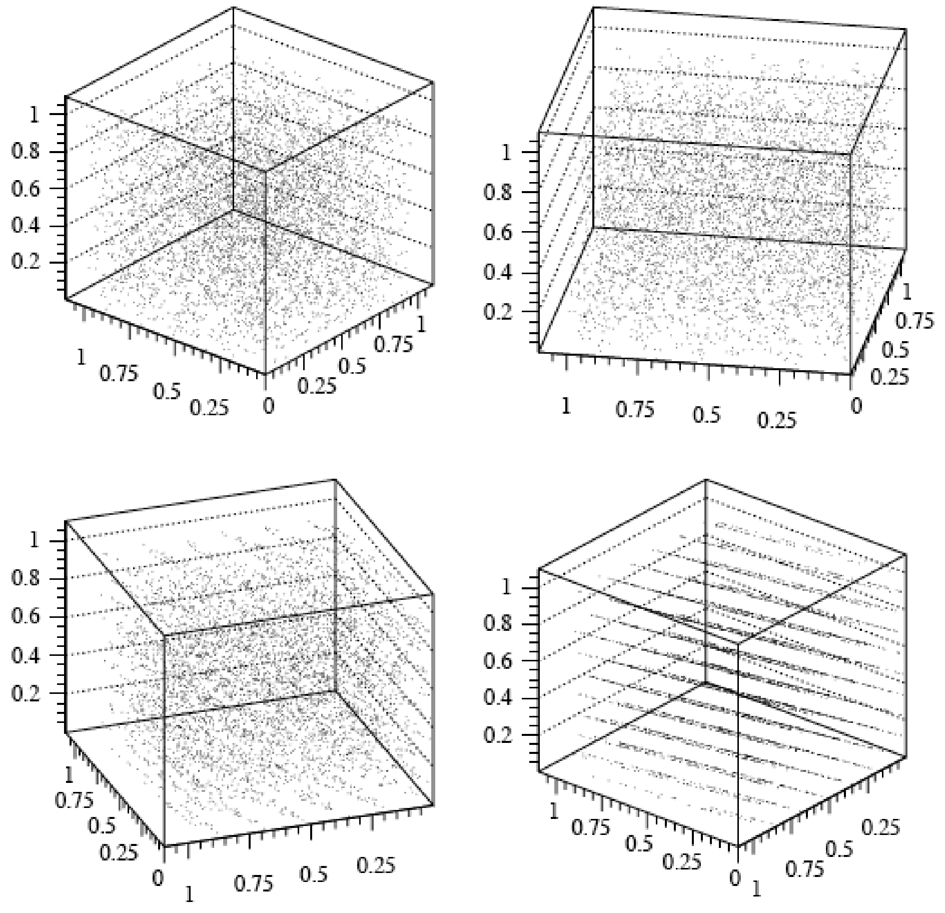
2D distribution Still looks okay $\longrightarrow$

plot pairs: $\mathbf{(x_i, x_{i+1})}$

## Results from Randu: 3D distribution

plot triplets:

$(x_i, x_{i+1}, x_{i+2})$



Problem seen when observed at the right angle!

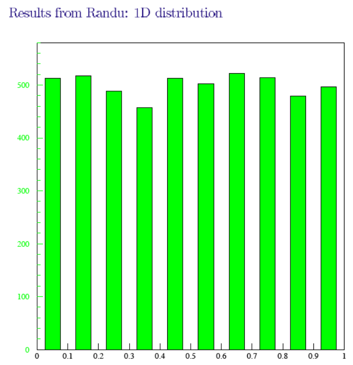
*Random numbers fall mainly in the planes*

Why? Hint: show that: $x_{k+2}=6x_{k+1}-9x_k$

other comments/references on: https://en.wikipedia.org/wiki/Linear_congruential_generator

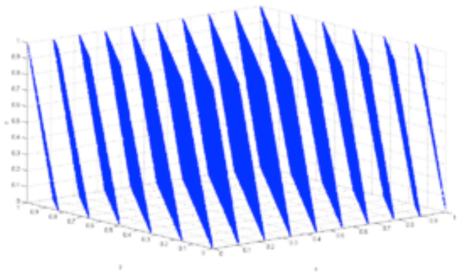
Also an example of Python code

# Tests the "quality" of a random sequence



- uniformity

(look at the histogram, but also check the moments of the distribution, i.e., $\langle x^k \rangle$, for k=1, 2, ...)



- correlation

- other more sophisticated tests

(in particular for cryptographically secure use!)

## true vs pseudo random number generators

| | **PSEUDO** | **TRUE** |
|---|---|---|
| efficiency | excellent | poor |
| determinism | deterministic | non deterministic |
| periodicity | periodic | aperiodic |

## Technicalities to create our own (pseudo)random number generator

mod ???

# Intrinsic procedures in FORTRAN

(see the page from Fortran90/95 for Scientists and Engineers, by S.J. Chapman)

`MOD(A1,P)`

- Elemental function of same kind as its arguments
- Returns the value `MOD(A,P) = A - P*INT(A/P)` if P ≠ 0. Results are processor dependent if P = 0.
- Arguments may be Real or Integer; they must be of the same type
- Examples:

| Function | Result |
|---|---|
| `MOD(5,3)` | 2 |
| `MOD(-5,3)` | -2 |
| `MOD(5,-3)` | 2 |
| `MOD(-5,-3)` | -2 |

`MODULO(A1,P)`

- Elemental function of same kind as its arguments
- Returns the modulo of A with respect to P if P ≠ 0. Results are processor dependent if P = 0.
- Arguments may be Real or Integer; they must be of the same type
- If P > 0, then the function determines the positive difference between A and then next lowest multiple of P. If P < 0, then the function determines the negative difference between A and then next highest multiple of P.
- Results agree with the MOD function for two positive or two negative arguments; results disagree for arguments of mixed signs.
- Examples:

| Function | Result | Explanation |
|---|---|---|
| `MODULO(5,3)` | 2 | 5 is 2 up from 3 |
| `MODULO(-5,3)` | 1 | -5 is 1 up from -6 |
| `MODULO(5,-3)` | -1 | 5 is 1 down from 6 |
| `MODULO(-5,-3)` | -2 | -5 is 2 down from -3 |

mod or modulo give the same result if acting on positive integers

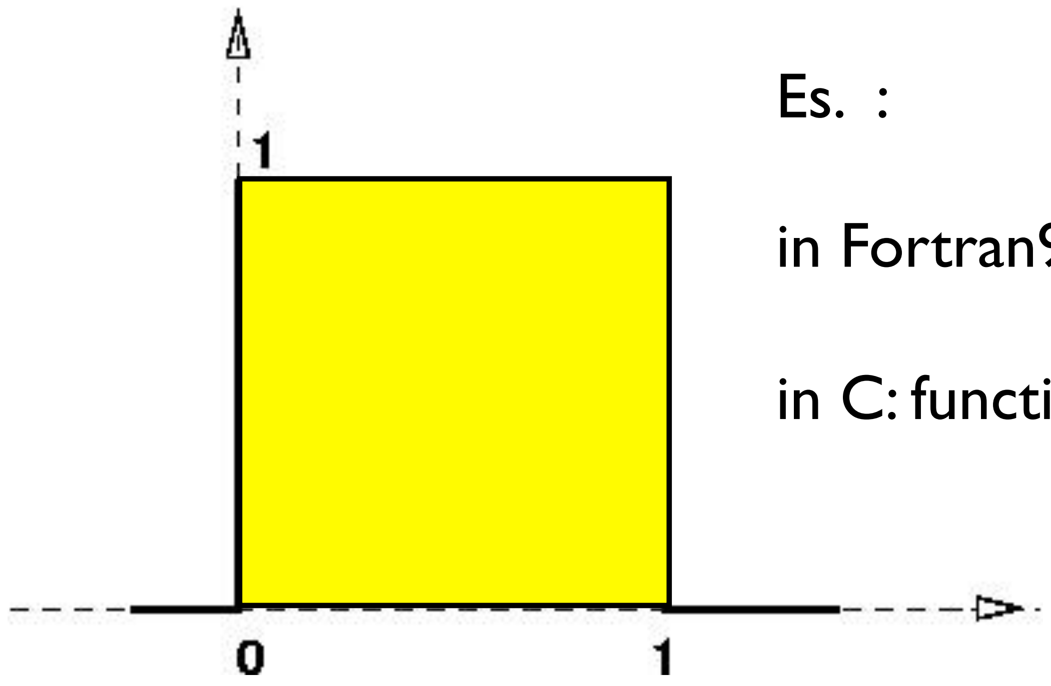
## Modulus operator in C++

the language provides a built-in mechanism, the **modulus operator** ('%'). Example:

```
#include <iostream>
using namespace std;

int main()
{
     int M = 8;
     int a = 5;
     int c = 3;
     int X = 1;
     int i;
     for(i=0; i<8; i++)
     {
          X = (a * X + c) % M;
          cout << X << " ";
     }
     return 0;
}
```

## Intrinsic pseudorandom numbers generators

We could create our own random number generator using "mod" intrinsic function, but it is much better to use directly the (smart) intrinsic procedures provided by the compilers to generate random numbers, in general: real, with uniform distribution in [0;1[



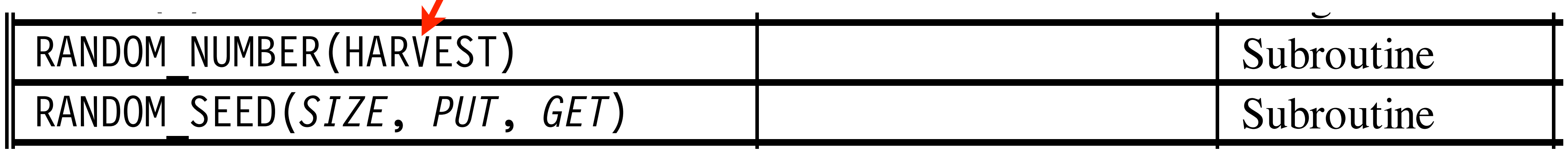
Es. :

in Fortran90: subroutine **random_number( )**

in C: function **rand** ...

# Intrinsic pseudorandom numbers generator in FORTRAN

the name of the produced output has to be specified



| | | |
|---|---|---|
| RANDOM_NUMBER(HARVEST) | | Subroutine |
| RANDOM_SEED(*SIZE*, *PUT*, *GET*) | | Subroutine |

*Here (Chapman's book): ARGUMENTS in Italic are* ***optional***
*(in other books, optional arguments are in square brackets [])*

## RANDOM_NUMBER(HARVEST)

- Intrinsic subroutine
- Returns pseudo-random number(s) from a uniform distribution in the range 0 ≤ HARVEST < 1. HARVEST may be either a scalar or an array. If it is an array, then a separate random number will be returned in each element of the array.
- Arguments:

| **Keyword** | **Type** | **Intent** | **Description** |
|---|---|---|---|
| HARVEST | Real | OUT | Holds random numbers. May be scalar or array. |

## RANDOM_SEED(*SIZE*,*PUT*,*GET*)

- Intrinsic subroutine
- Performs three functions: (1) restarts the pseudo-random number generator used by subroutine RANDOM_NUMBER, (2) gets information about the generator, and (3) puts a new seed into the generator.
- Arguments:

| **Keyword** | **Type** | **Intent** | **Description** |
|---|---|---|---|
| *SIZE* | Integer | OUT | Number of integers used to hold the seed ($n$) |
| *PUT* | Integer($m$) | IN | Set the seed to the value in *PUT*. Note that $m \geq n$. |
| *GET* | Integer($m$) | OUT | Get the current value of the seed. Note that $m \geq n$. |

- *SIZE* is an Integer, and *PUT* and *GET* are Integer arrays. All arguments are optional, and at most one can be specified in any given call.
- Functions:
  1. If no argument is specified, the call to RANDOM_SEED restarts the pseudo-random number generator.
  2. If *SIZE* is specified, then the subroutine returns the number of integers used by the generator to hold the seed.
  3. If *GET* is specified, then the current random generator seed is returned to the user. The integer array associated with keyword *GET* must be at least as long as *SIZE*.
  4. If *PUT* is specified, then the value in the integer array associated with keyword *PUT* is set into the generator as a new seed. The integer array associated with keyword *PUT* must be at least as long as *SIZE*.

warning:
processor-
dependent;
sometimes it
starts always
from
the same
seed !!!

# Intrinsic pseudorandom numbers generator in FORTRAN

subroutine **random_number(x)** :
- the argument x can be either a scalar or a N-dimensional array
- the result is one or N ***real pseudorandom numbers*** uniformly distributed between 0 and 1

subroutine **random_seed([size][put] [get])**
- algorithm is deterministic: the sequence can be controlled by initialization: array of “size” (*) integers (***seed***): different ***seeds*** -> different sequences
- syntax:

***call random_seed(put=seed)*** to put seed,
***call random_seed(get=seed)*** to get its value

(*): it depends on the compiler (gfortran, g95, ifort, ...) and on the machine architecture

# Intrinsic pseudorandom numbers generator in FORTRAN

Further notes:

subroutine **random_number(x)** :
- you can call it directly, without a previous call to random_seed

subroutine **random_seed([size][put][get])**
- all the arguments are optional; i.e., you may also call it as:
call random_seed()
The call without arguments corresponds to different actions, according to the compiler implementation and is processor dependent!!! **check** on your computer!
In some cases it starts always from the same seed, chosen by the computer

## Intrinsic pseudorandom numbers generator in C++

***real pseudorandom numbers*** uniformly distributed between 0 and 1:

**temp = rand();**

A number between 0 and 50:

**int rnd = int((double(rand())/RAND_MAX)*50);**

where RAND_MAX is an implementation defined constant.

Also in c++ the sequence can be controlled by initialization:

**srand ( time(NULL) );**

## Some programs:

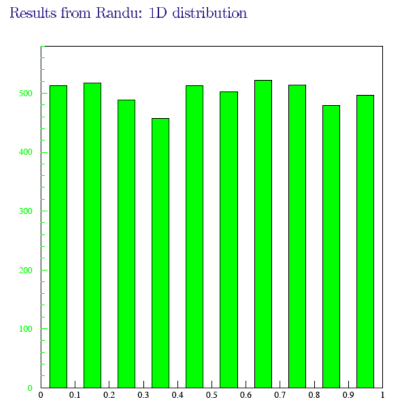
on moodle2.units.it

random_lc.f90
rantest_intrinsic.f90
rantest_intrinsic_with_seed.f90
rantestbis_intrinsic.f90
INIT_RANDOM_SEED.f90
nrdemo_ran.f90

## Exercise 1a:
## test of **uniformity** of the pseudorandom sequence

r(n), n=1, data is our random number sequence between 0 and 1

(b) Do a histogram with the sequence generated above and plot it using for instance **gnuplot** with the command `w[ith] boxes`. Is the distribution *uniform*?
*Hint: to do the histogram, divide the range into a given number of channels of width $\Delta r$, then calculate how many points fall in each channel, $r/\Delta r$:*



```
integer, dimension(20) :: histog
 :
histog = 0
do n = 1, ndata
   i = int(r(n)/delta_r) + 1
   histog(i) = histog(i) + 1
end do
```

<= counts the number of points falling between i*delta_r and (i+1)*delta_r and assign them to the "i+1" channel

# what is int() ? similar intrinsic functions? how to choose?

**AINT(A[,KIND])**
- Real elemental function
- Returns A truncated to a whole number. AINT(A) is the largest integer which is smaller than |A|, with the sign of A. For example, AINT(3.7) is 3.0, and AINT(-3.7) is -3.0.
- Argument A is Real; optional argument KIND is Integer

**ANINT(A[,KIND])**
- Real elemental function
- Returns the nearest whole number to A. For example, ANINT(3.7) is 4.0, and AINT(-3.7) is -4.0.
- Argument A is Real; optional argument KIND is Integer

**FLOOR(A,KIND)**
- Integer elemental function
- Returns the largest integer ≤ A. For example, FLOOR(3.7) is 3, and FLOOR(-3.7) is -4.
- Argument A is Real of any kind; optional argument KIND is Integer
- Argument KIND is only available in Fortran 95

**INT(A[,KIND])**
- Integer elemental function
- This function truncates A and converts it into an integer. If A is complex, only the real part is converted. If A is integer, this function changes the kind only.
- A is numeric; optional argument KIND is Integer.

**NINT(A[,KIND])**
- Integer elemental function
- Returns the nearest integer to the real value A.
- A is Real

## what is int() ? similar intrinsic functions? how to choose?

**AINT(A[,KIND])**
· Real elemental function
· Returns A truncated to a whole number. AINT(A) is the largest integer which is smaller than |A|, with the sign of A. For example, AINT(3.7) is 3.0, and AINT(-3.7) is -3.0.
· Argument A is Real; optional argument KIND is Integer

**ANINT(A[,KIND])**
· Real elemental function
· Returns the nearest whole number to A. For example, ANINT(3.7) is 4.0, and AINT(-3.7) is -4.0.
· Argument A is Real; optional argument KIND is Integer

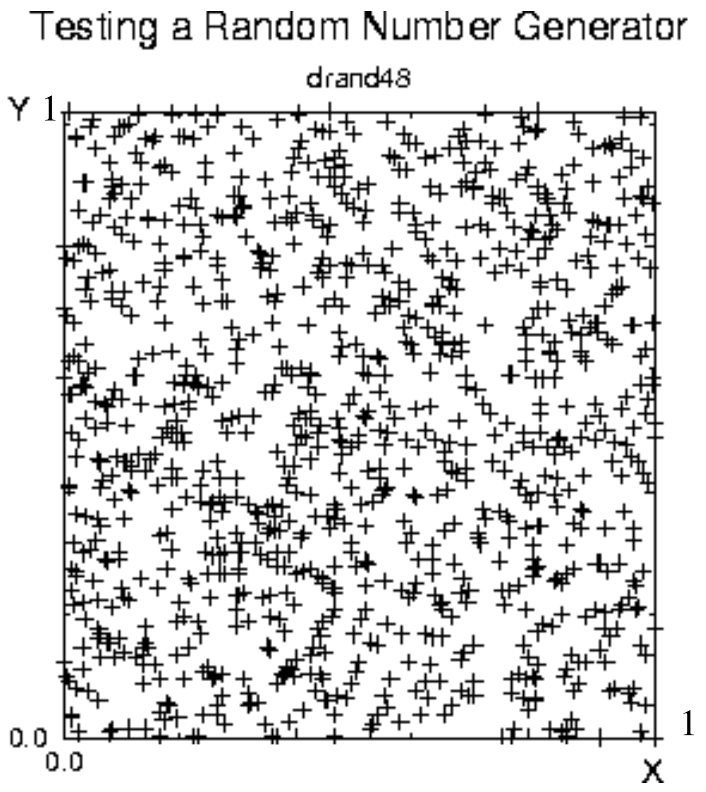
**FLOOR(A,KIND)**
· Integer elemental function
· Returns the largest integer ≤ A. For example, FLOOR(3.7) is 3, and FLOOR(-3.7) is -4.
· Argument A is Real of any kind; optional argument KIND is Integer
· Argument KIND is only available in Fortran 95

**INT(A[,KIND])**
· Integer elemental function
· This function truncates A and converts it into an integer. If A is complex, only the real part is converted. If A is integer, this function changes the kind only.
· A is numeric; optional argument KIND is Integer.

**NINT(A[,KIND])**
· Integer elemental function
· Returns the nearest integer to the real value A.
· A is Real

## Exercise 1b:
## intrinsic random number generator - test **correlations**

$$(x_i, y_i) = (r_{2i-1}, r_{2i}) \qquad i = 1, 2, 3....$$



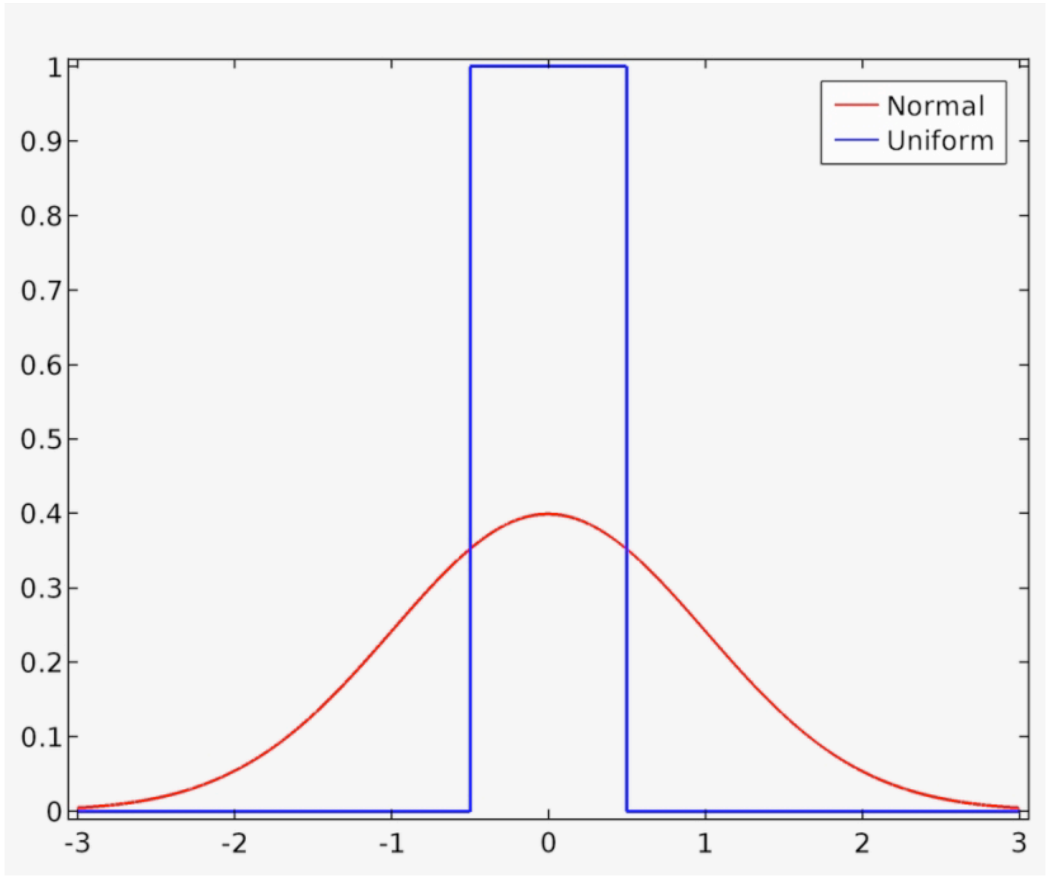
( obsolete: fortran 77 )

How many numbers? How many pairs?

## Exercise 2:
## intrinsic random number generator - test **uniformity**

## Quantitative tests the "quality" of a random sequence

two distributions are the same if all the moments $<x^k>$ are the same, and not just the first one $<x^1>$ (average)



e.g.:

uniform and gaussian distribution centred around zero have the same average, but different higher order momenta

# Exercise 2: intrinsic random number generator - test **uniformity**

(a) For a *uniformity* quantitative test, calculate the moment of order $k$:

$$\langle x^k \rangle^{calc} = \frac{1}{N} \sum_{i=1}^{N} x_i^k ,$$

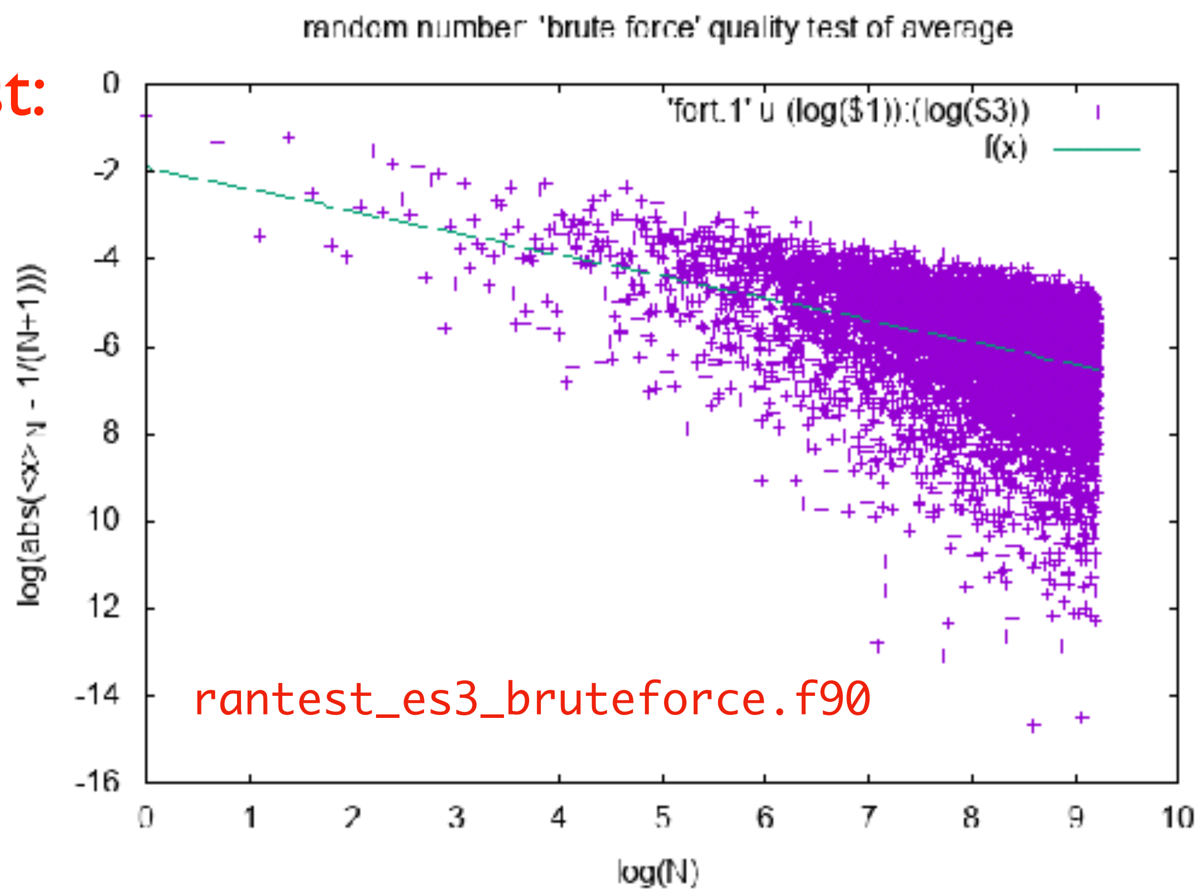
that should correspond to

$$\langle x^k \rangle^{th} = \int_0^1 dx\ x^k\ p_u(x) = \frac{1}{k+1}$$

where $p_u(x)$ is the uniform distribution in $[0,1[$. For a given $k$ (fix for instance $k$=1, 3, 7), consider the deviation of the calculated momentum from the expected one: $\Delta_N(k) = \left| \langle x^k \rangle^{calc} - \langle x^k \rangle^{th} \right|$, and study its behaviour with $N$ ($N$ up to ~100.000). It should be $\sim 1/\sqrt{N}$. *(a log-log plot could be useful)*

$$\text{If } f(x) \sim 1/\sqrt{N} \Longrightarrow log(f(x)) \sim -\frac{1}{2}\ log(N)$$

## A “brute force” test: Do several sequences of different length



rantest_es3_bruteforce.f90

```
…
do i=1,N
allocate (rnd(i))

call random_number(rnd)  ! generate a new sequence of "i" random numbers
                         ! (seed changes automatically)
somma = sum(rnd**k)   <= this sum() corresponds to an internal loop (nested loops)
write(1,*)i,somma/i, abs(somma/i - 1./(k+1))
! somma/i is the PARTIAL sum of the sequence for the momentum k
deallocate(rnd)

end do
```

ok, but time consuming…

## how to calculate the sum of the series for increasing N?

no need of recalculating again the sum from scratch; print out **partial** sums:

rantest_es3_simplest.f90

```
implicit none
integer :: N, i, k
real :: sum
real, dimension (:), allocatable :: rnd

print*,' Insert how many random numbers >'
read(*,*)N
allocate (rnd(N))
call random_number(rnd)

print*,' Insert the order of momentum >'
read(*,*)k

sum = 0.

open (unit=1,file='momentumk.dat')

do i=1,N
sum = sum + rnd(i)**k
write(1,*)i,sum/i, abs(sum/i - 1./(k+1))
! sum/i is the PARTIAL sum of the sequence for the momentum k
end do
```

print out the result as a function of "i"

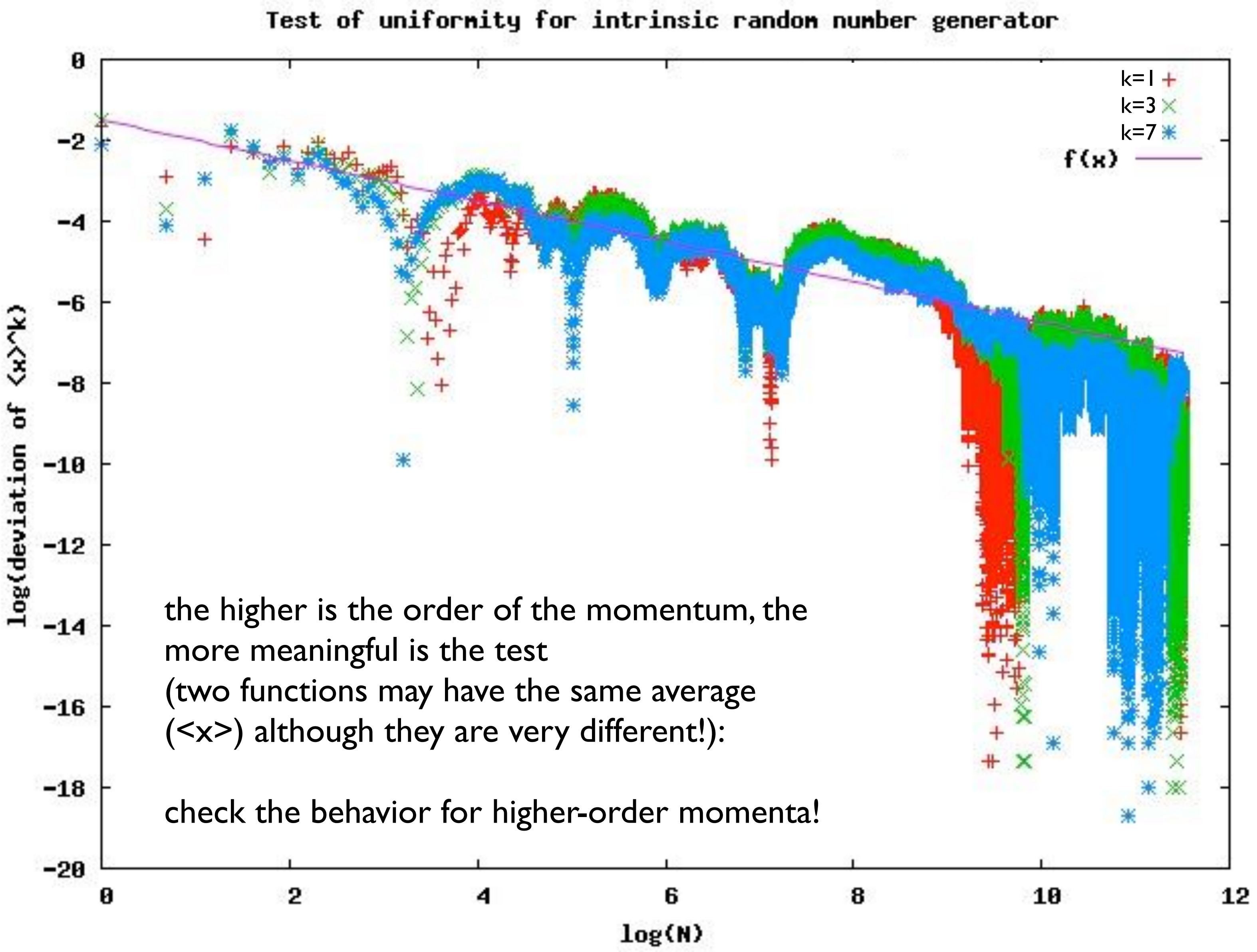
## Test on one sequence, several momenta

rantest_es3_simple.f90

```
…
allocate (rnd(N))
call random_number(rnd)
…
allocate(sum(kmax))
..
sum = 0.

do k = 1, kmax  ! Loop for the different momenta

do i=1,N
sum(k) = sum(k) + rnd(i)**k
write(klabel,*)i, sum(k)/i, abs(sum(k)/i - 1./(k+1))
! sum(k)/i is the PARTIAL sum of the sequence for the momentum k
end do ! I

close(klabel)
end do ! k
```

$\longrightarrow$ also here print the results as a function of “i”

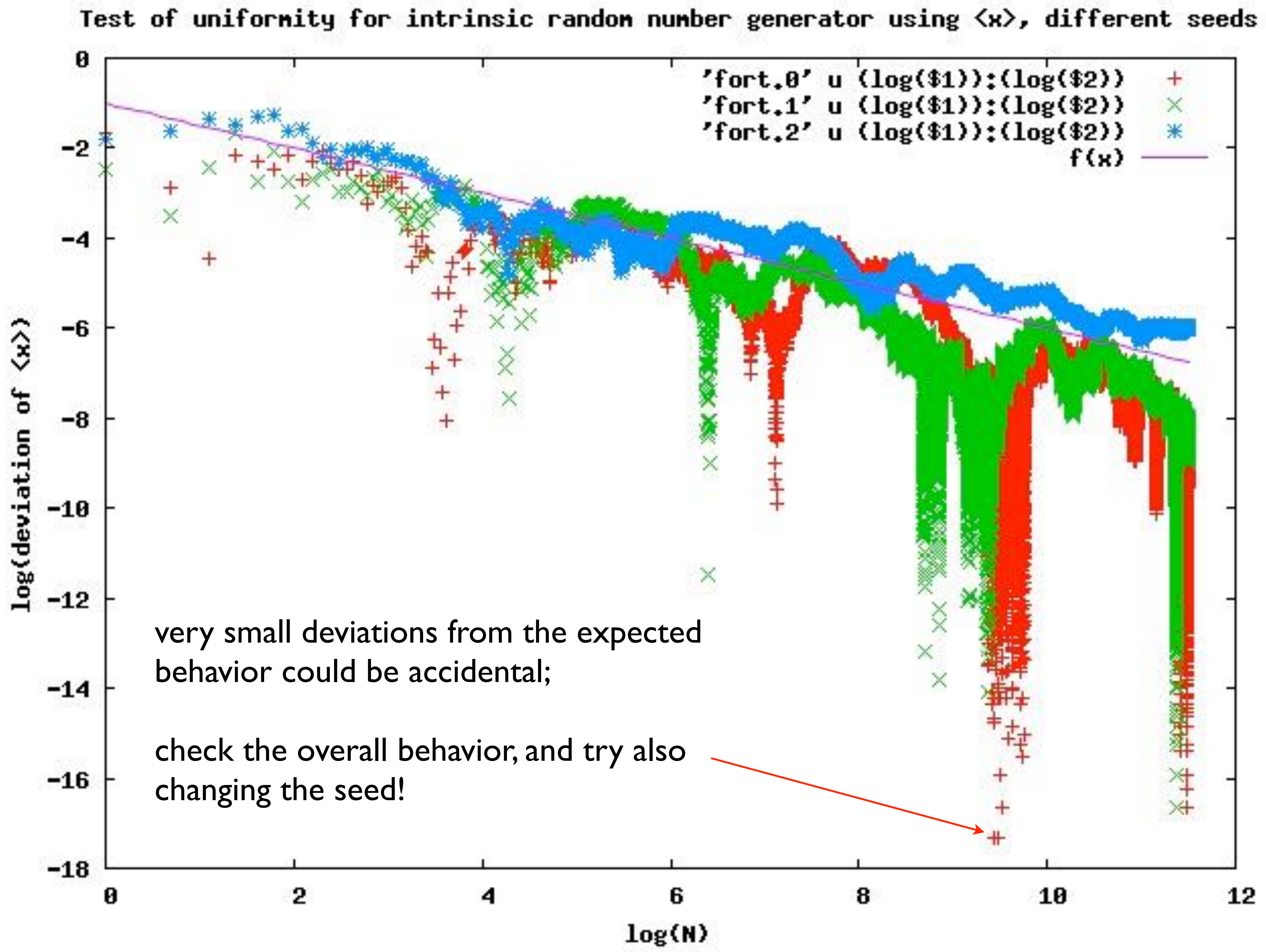
## Test on one sequence, several momenta



the higher is the order of the momentum, the more meaningful is the test
(two functions may have the same average (<x>) although they are very different!):

check the behavior for higher-order momenta!

## Test on different sequences for a given momentum



very small deviations from the expected behavior could be accidental;

check the overall behavior, and try also changing the seed!

## A general suggestion:

## do you want to check a power law?

$$\text{deviation of } \langle x \rangle^k = \left| \frac{1}{N}\sum_{i=1}^{N} x_i^k - \frac{1}{k+1} \right| \sim 1/\sqrt{N} + \text{cost.}$$

numerically calculated

from the sequence

expected if the sequence

was

truly uniform

$\downarrow$

## linear regression: much better

log(deviation of <x>$^k$) ~ -1/2 log(N) + cost.'

check the slope of the log-log !!!

## do you want to fit with gnuplot?

Suppose you have the data in two columns, x and y, and you suspect a power low $y = x^a$ + const

Consider that: $\log(y) = a * \log(x) + b$

```
gnuplot> f(x) = a * x + b

gnuplot> fit f(x) 'data.dat' u (log($1)):(log($2)) via a,b

gnuplot> plot f(x), 'data.dat'
```

# Exercise 3 [specific for FORTRAN90]: use of the seed

```
integer, dimension(:), allocatable :: seed
integer :: sizer
...
call random_seed(sizer)
! the result depends of the machine architecture!

allocate(seed(sizer))
```

## A few instructions/suggestions specifically for Fortran90

## main program & modules

You can:
prepare a module: *modulename.f90*
prepare the main code that uses the module: *mainprogram.f90*
then:

Compile the module with the option -c: this produces .mod and .o (the objects):

```
gfortran -c modulename.f90
```

Compile the main program:

```
gfortran -c mainprogram.f90
```

Finally you link all the files **.o* and produce the executable:

```
gfortran -o a.out mainprogram.o modulename.o
```

## Data input / output

you can:
prepare an input datafile (say, in.dat)

then:

```
$ ./a.out < in.dat
```

Also the output can be redirected:

```
$ ./a.out > out.dat
```

*(Note: EVERYTHING is redirected!*
*You won’t see anything on the screen)*