993SM - Laboratory of Computational Physics
IV week
October 16, 2023

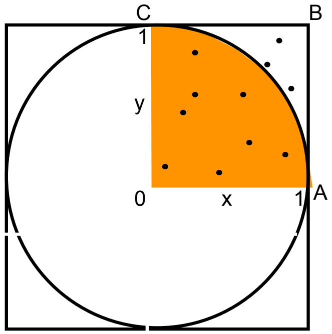
Maria Peressi
Università degli Studi di Trieste - Dipartimento di Fisica
Sede di Miramare (Strada Costiera 11, Trieste)
e-mail: peressi@units.it
tel.: +39 040 2240242

Random numbers and Monte Carlo(*) Techniques

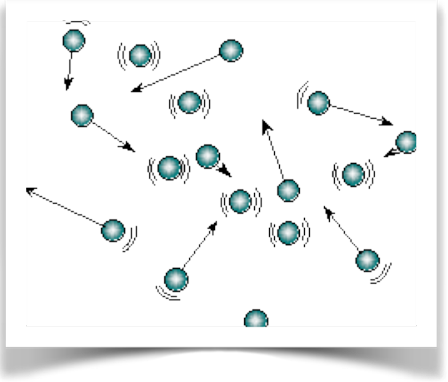
(*) any procedure making use of random numbers

Random numbers: use

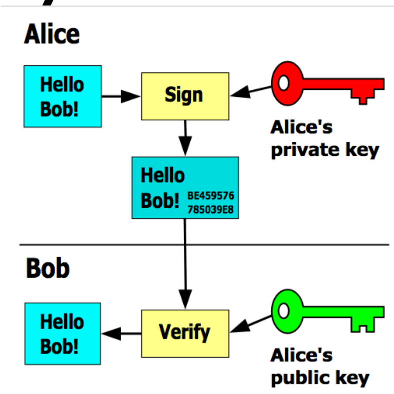
- in numerical analysis (to calculate integrals)



- to simulate and model complex or intrinsically random phenomena



- to generate data encryption keys



- ...

Random numbers:

Characteristics and Generation

Random numbers

A sequence of random numbers is a set of numbers that have nothing to do with the other numbers in the sequence.

... but with a well defined statistical properties, e.g.:

In a uniform distribution of random numbers in the range [0,1] , every number has the same chance of turning up.

Note that 0.00001 is just as likely as 0.50000

True random numbers generation

- Use some chaotic systems, like numbered balls in a barrel (Lotto game)

- Use a process that is inherently random, such as:

  - radioactive decay
  - thermal noise
  - cosmic ray arrival

- Tables of a few million truly random numbers do exist, but this is not enough for most scientific applications

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 !!!]

INTEGER
(pseudo)Random numbers generation

many different algorithms...

Two among the simplest (and oldest) algorithms:
- von Neumann
- Linear Congruential Method

(pseudo)random numbers generation:
example 1 - “Middle square” algorithm

(Von Neumann, 1946)

To generate a 10-digit integer sequence:

- take a first one
- square it
- consider the 10 central digits

loop

eg. $5772156649^2=33317\underline{7923805949}09291$

so the next number is given by

Also this sequence may “appear” to be random.

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

depending on the initial choice, you can be trapped into short loops:

$6100^2=37210000$
$2100^2=\ \ 4410000$
$4100^2=16810000$
$8100^2=65610000$

(pseudo)random numbers generation:
example II - "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

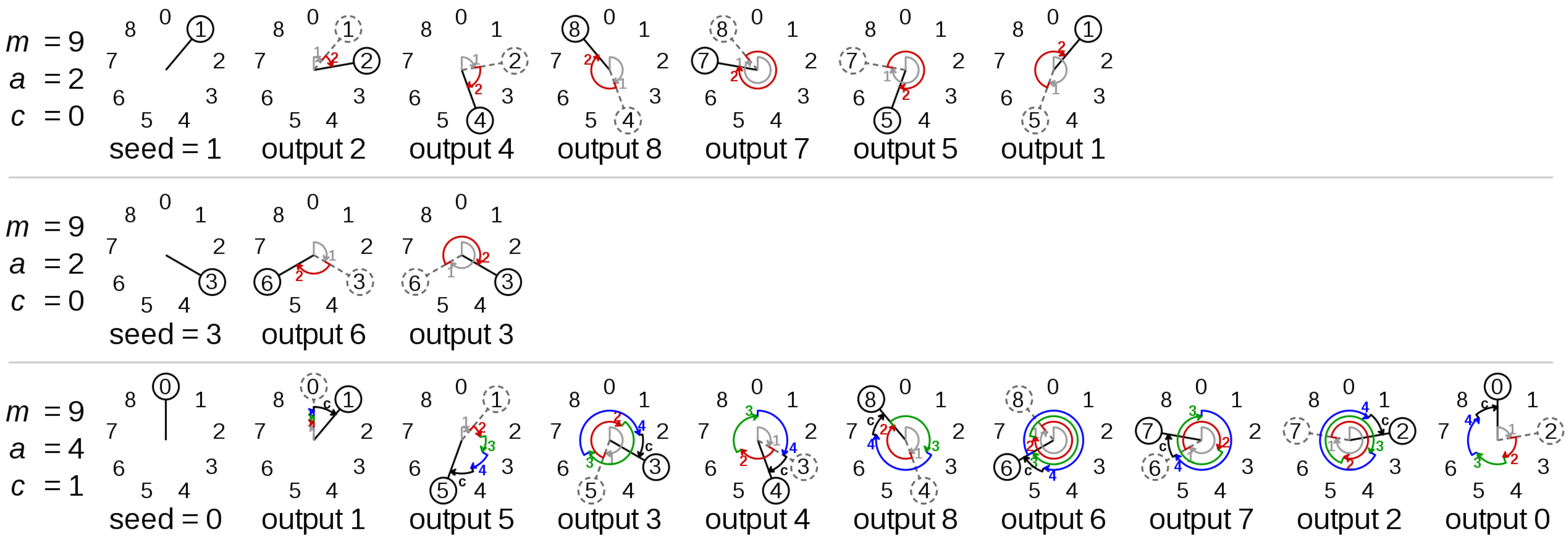
(pseudo)random numbers generation:
example II - "Linear congruential method (LCM)"

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

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?

(pseudo)random numbers generation:
example II - “Linear congruential method (LCM)”



https://commons.wikimedia.org/wiki/File:Linear_congruential_generator_visualisation.svg

Limits of the algorithm:

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

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

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 $2^{31} \approx 2\times10^{9}$.

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...

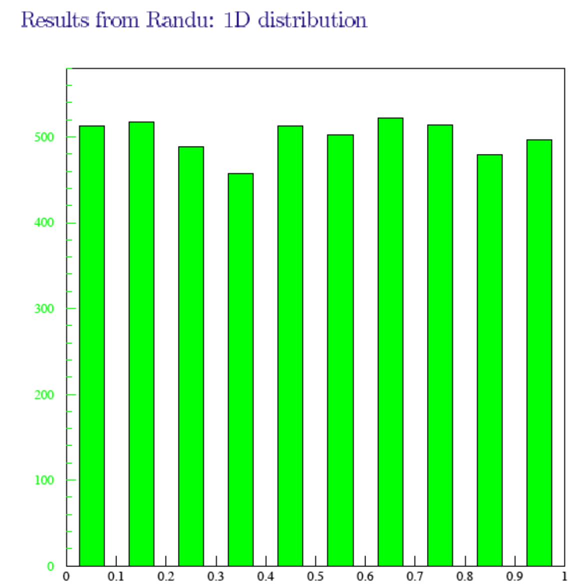
A popular random number generator was distributed by IBM in the 1960's with the algorithm: https://en.wikipedia.org/wiki/RANDU

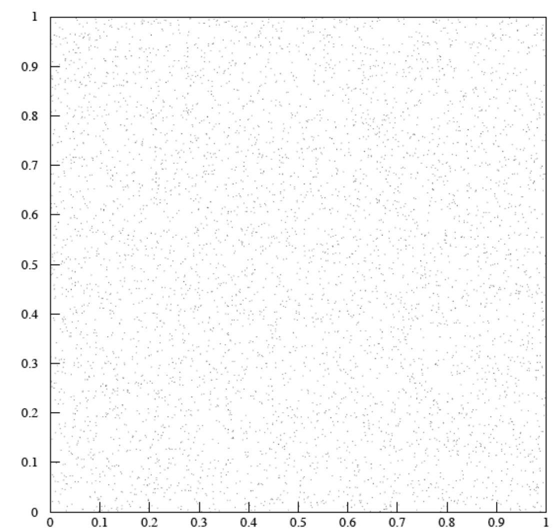
$$I_{n+1} = (65539 \times I_n) \bmod 2^{31}$$

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

1D distribution Looks okay

$x_i, p(x_i)$





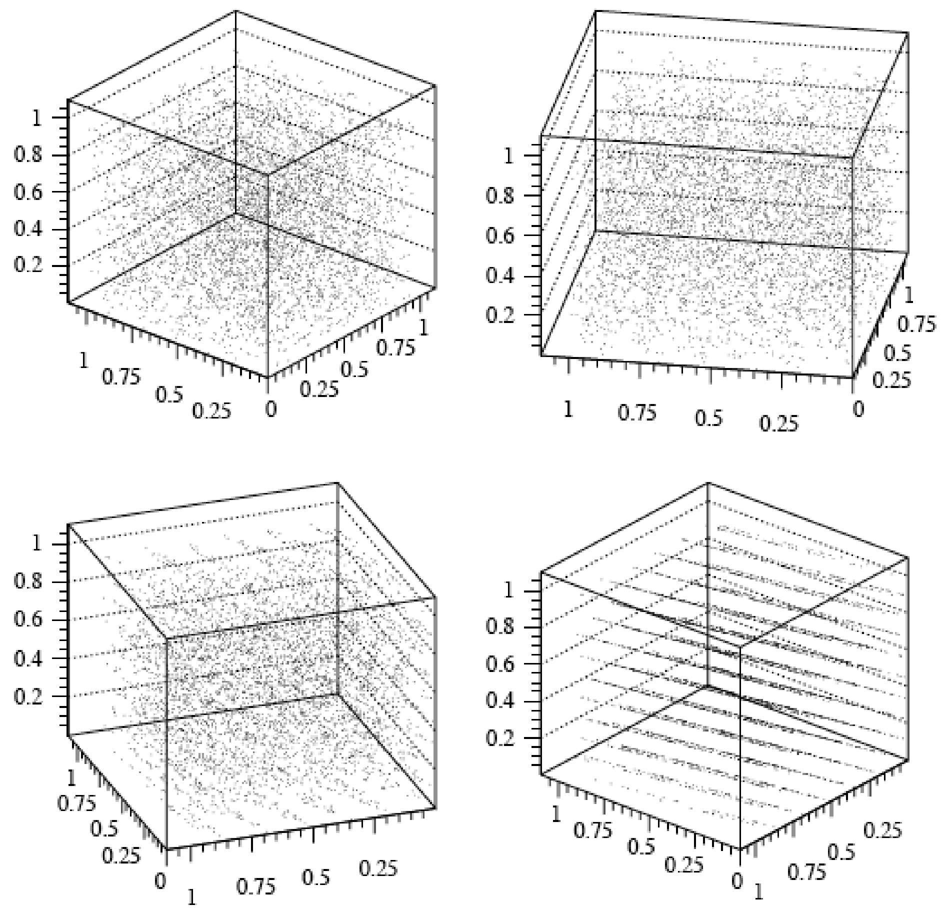
2D distribution Still looks okay

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

plot triplets:

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

Results from Randu: 3D distribution



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

Problems also with other smart algorithms ...

The authors of *Numerical Recipies* have admitted that the random number generators, RAN1 and RAN2 given in the first edition, are "at best mediocre".

In their second edition, these are replaced by ran0, ran1, and ran2, which have much better properties.

many editions, see **web site: numerical.recipes**;
free old edition (1996) in fortran:
http://s3.amazonaws.com/nrbook.com/book_F210.html =>
II edition in Fortran90 => B7 Random Numbers p. 1141 or
http://nrbook.com/a/bookf90pdf.php => Random number in Ch. 7
(you need the FileOpen plugin for Adobe [Acrobat] Reader®)

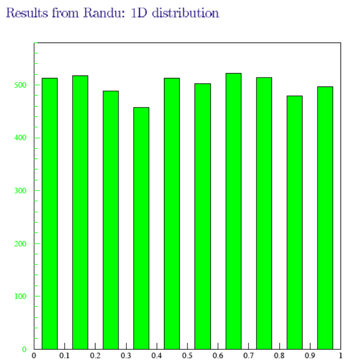
Possible improvements

One way to improve the behaviour of random number generators and to increase their period is to modify the algorithm:

$$I_n = (a \times I_{n-1} + b \times I_{n-2}) \bmod m$$

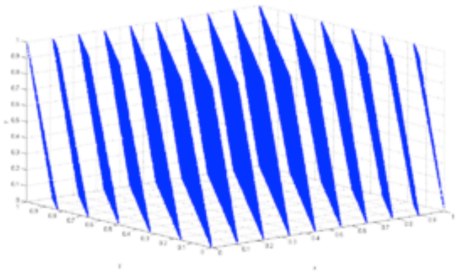
Which in this case has two initial seeds and can have a period greater than m.

Tests the "quality" of a random sequence



- uniformity

(look at the histogram, but also check the moments of the distribution, i.e., <x^k>, for k=1, 2, ...)



- correlation

- other more sophisticated tests

(in particular for cryptographically secure use!)

Many other (pseudo)random numbers generators

- "Mersenne twister" (Matsumoto and Nishimura , 1997)

The commonly used variant, MT19937, produces a sequence of 32-bit integers with the following desirable properties:

1. It has a very long period of $2^{19937} - 1$ (which is necessary but not sufficient to guarantee of good quality in a random number generator)
2. It passes numerous tests for statistical randomness

- ...

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 reference to Chapman book on the moodle page on this Course)

Table B-1: Specific and Generic Names for All Fortran 90/95 Intrinsic Procedures

Generic name, keyword(s), and calling sequence	Specific name	Function type	Section	Notes
`ABS(A)`		Argument type	B.3	
	`ABS(r)`	Default real		
	`CABS(c)`	Default real		2
	`DABS(d)`	Double Prec.		
	`IABS(i)`	Default integer		
`ACHAR(I)`		Character(1)	B.7	
`ACOS(X)`		Argument type	B.3	
	`ACOS(r)`	Default Real		
	`DACOS(d)`	Double Prec.		
`ADJUSTL(STRING)`		Character	B.7	
`ADJUSTR(STRING)`		Character	B.7	
`AIMAG(Z)`	`AIMAG(c)`	Real	B.3	
`AINT(A, `*`KIND`*`)`		Argument type	B.3	
	`AINT(r)`	Default Real		
	`DINT(d)`	Double Prec.		

...

EXPANDED DESCRIPTION OF FORTRAN 90 / 95 INTRINSIC PROCEDURES

Intrinsic procedures in FORTRAN

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

`MOD(A,P)`	
	`AMOD(r1,r2)`
	`MOD(i,j)`
	`DMOD(d1,d2)`
		Argument type
		Real
		Integer
		Double Prec.
			B.3	

`MODULO(A,P)`		Argument type	B.3	

…

Intrinsic procedures in FORTRAN

`MOD(A1,P)`

- Elemental function of same kind as its arguments
- Returns the value `MOD(A,P) = A - P*INT(A/P)` if $P \neq 0$. 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 $P \neq 0$. 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

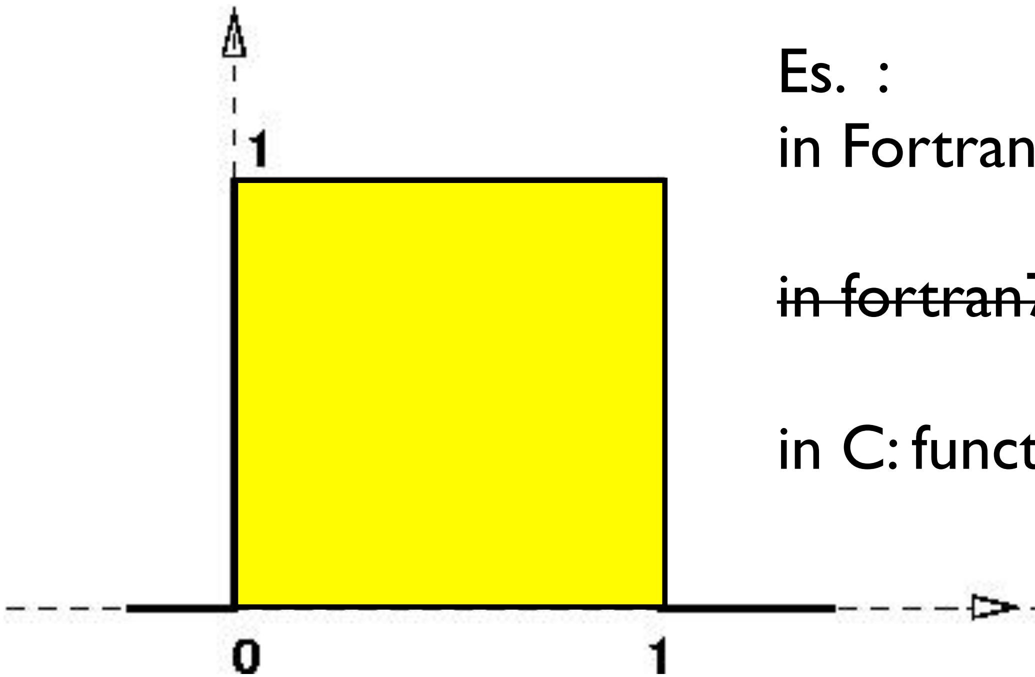
Modulus operator in C++

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 fortran77: function **drand48( )**~~

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:

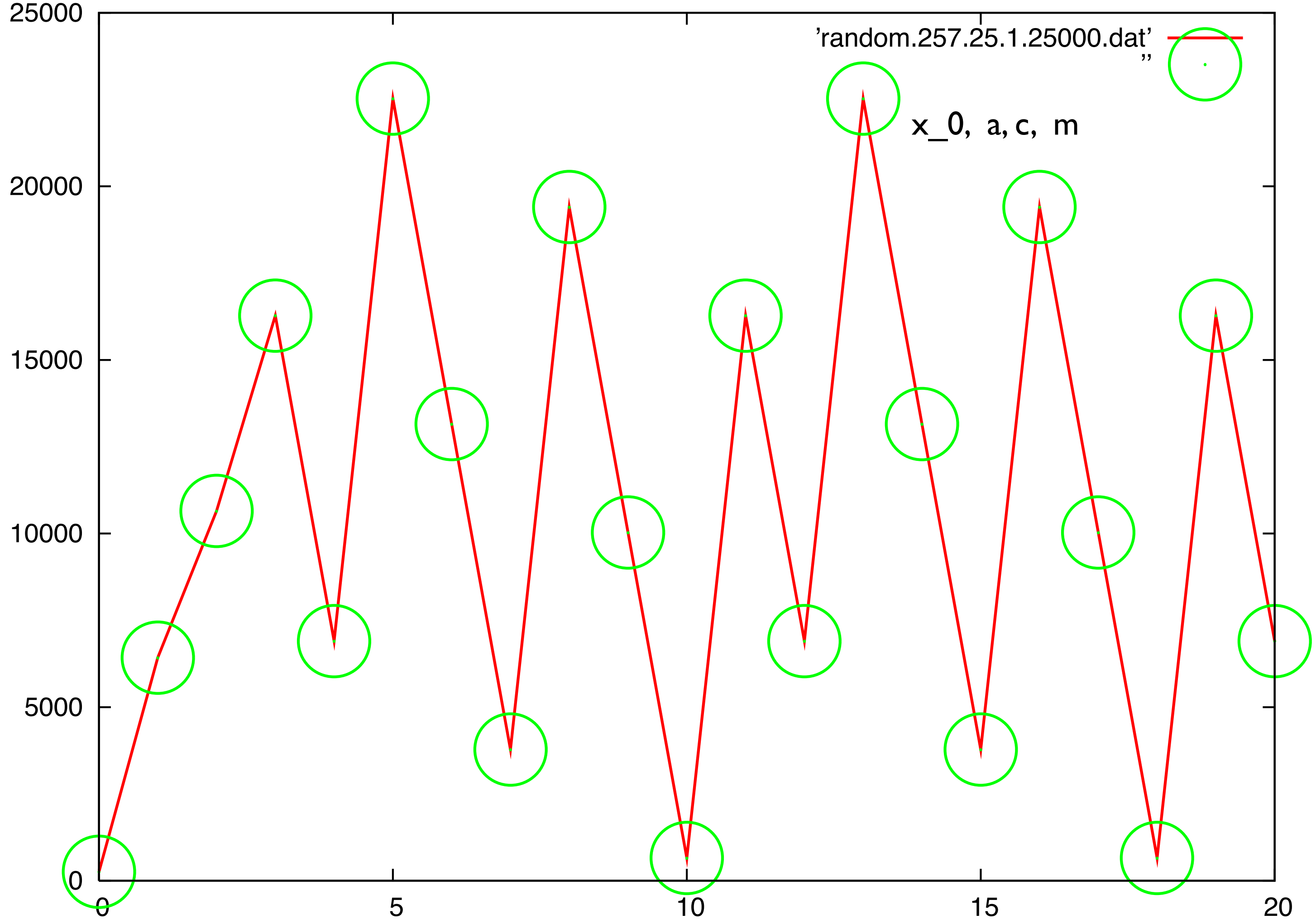
```
srand ( time(NULL) );
```

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 I:
Linear Congruent Method: **periodicity**



How to determine the period "automatically"?
Is it enough to check when a generated number is equal to the initial seed?
NO. In same cases you will NEVER go back to the seed...

A possible algorithm:

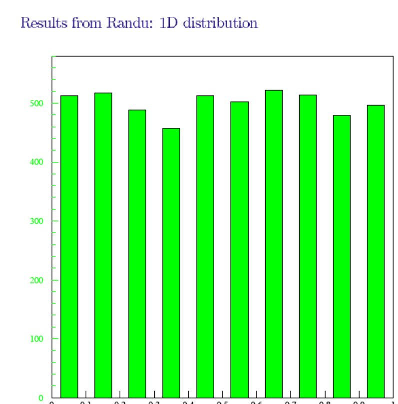
- create a sequence of $m+1$ numbers (you don't need more! why?)
- don't start from the first one, that could be in a transient part of the sequence, but from the last one, which is for sure in the periodic part
- compare all the numbers with the last one, starting from the second to the last and going back by 1 ...
- you get the period!

Exercise 2:
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 Δr, 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

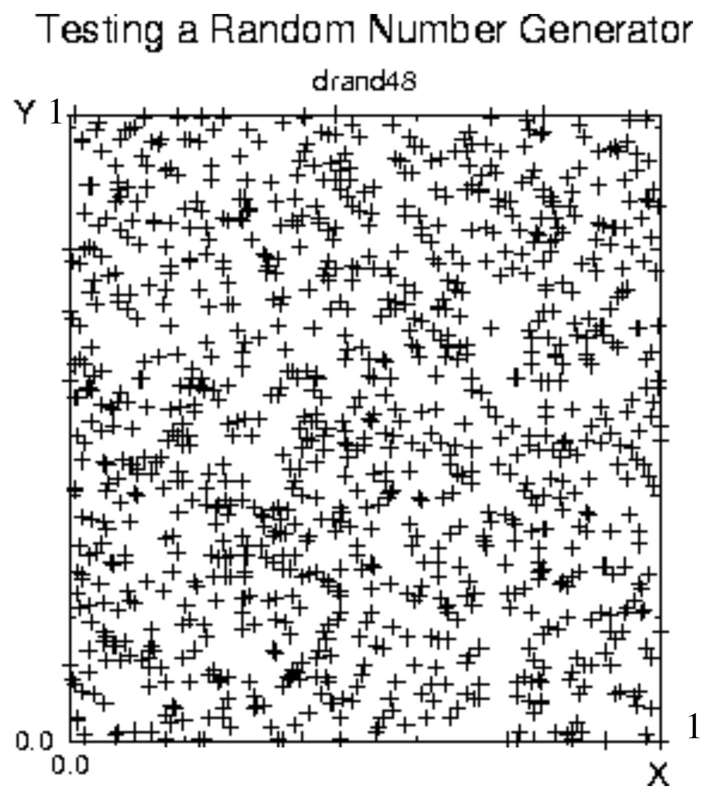
FLOOR(A,KIND)
· Integer elemental function
· Returns the largest integer ≤ A. For example, FLOOR(3.7) is 3, and FLOOR(-3.7) is -4.
· Argument A is Real of any kind; optional argument KIND is Integer
· Argument KIND is only available in Fortran 95

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 2:
intrinsic random number generator - test **correlations**

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



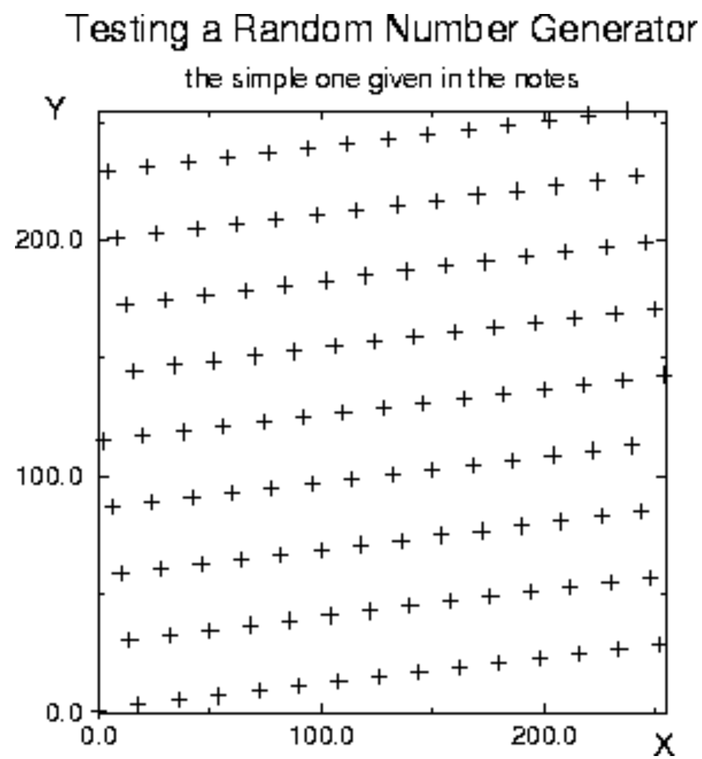
(obsolete: fortran 77)

How many numbers? How many pairs?

but...

correlations with the LCM generator with M=256

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

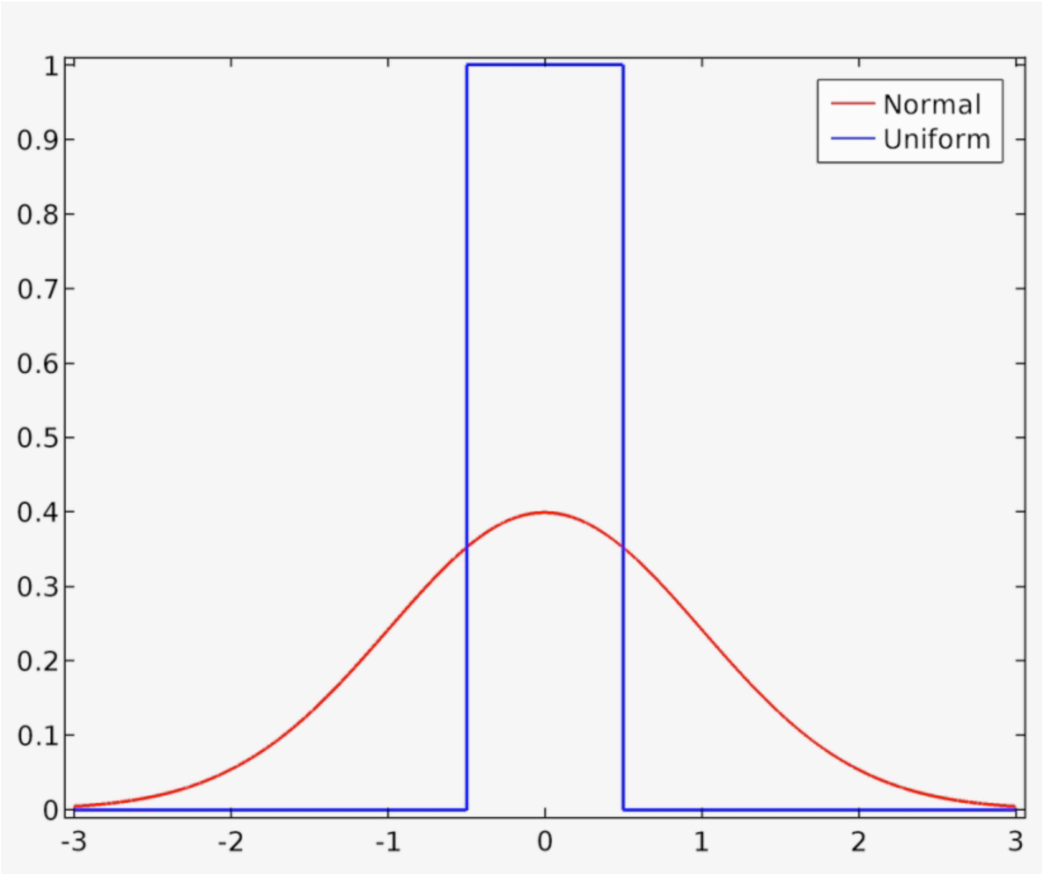


How many numbers? How many pairs?

Exercise 3:
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 3:
intrinsic random number generator - test **uniformity**

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

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

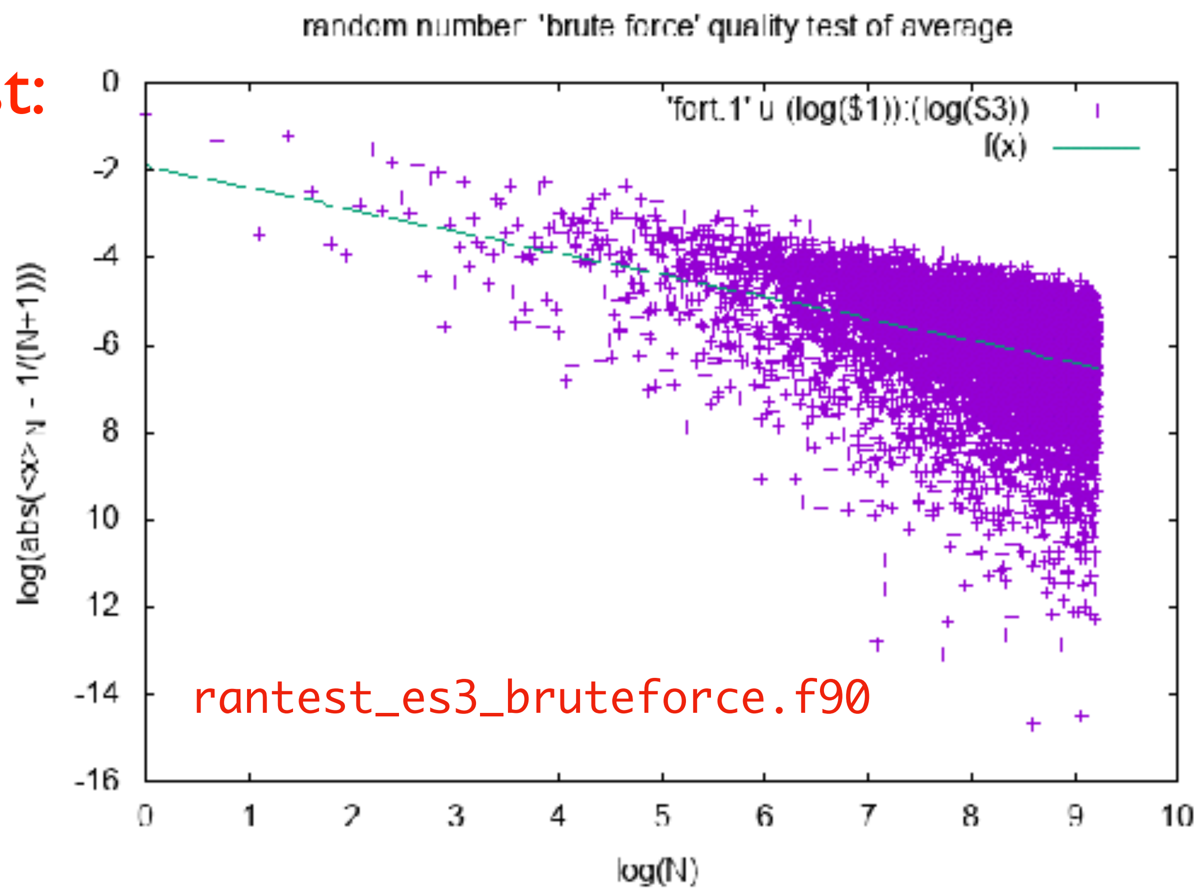
that should correspond to

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

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




```
…
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"

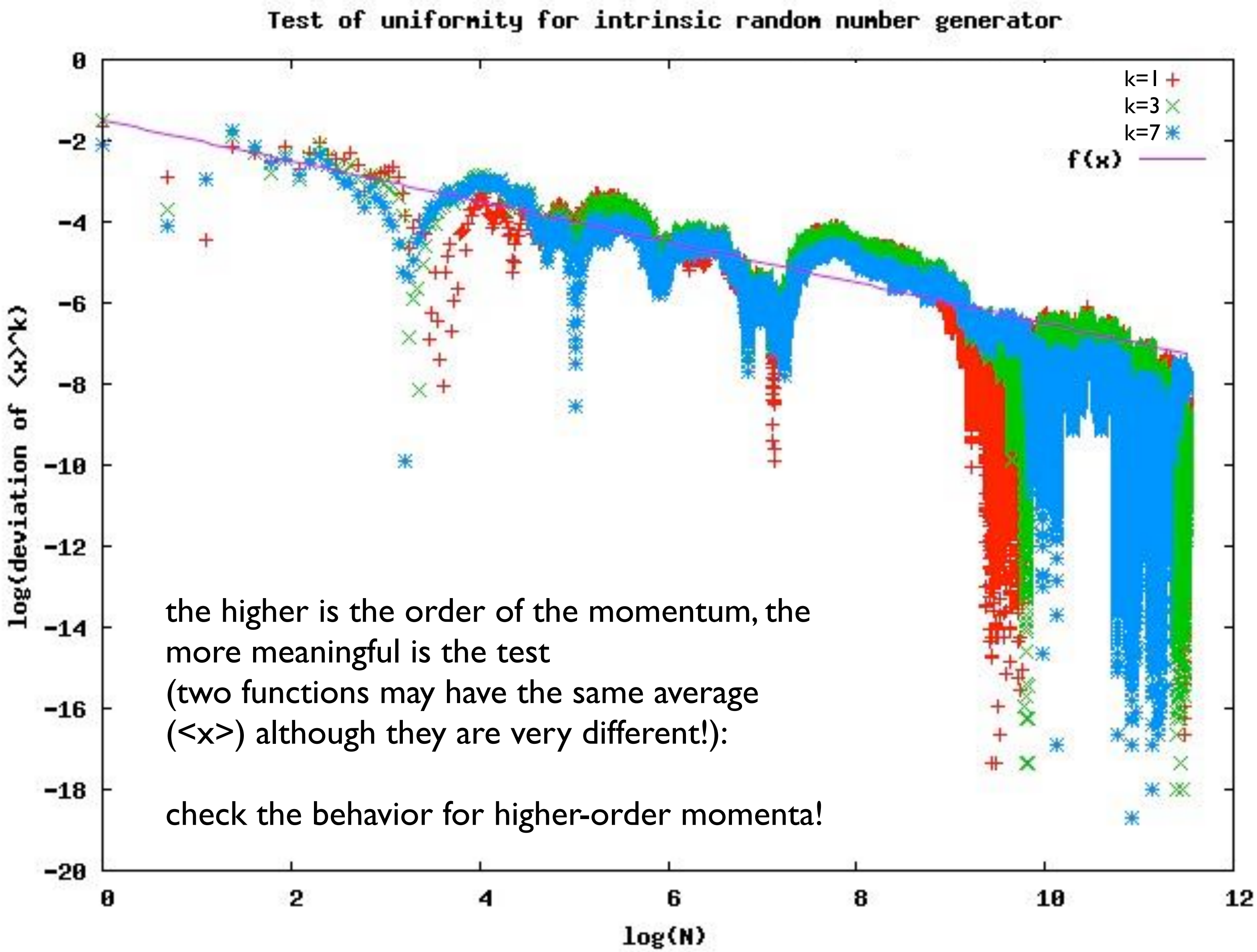
Test on one sequence, several momenta

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
```

also here print the results as a function of "i"

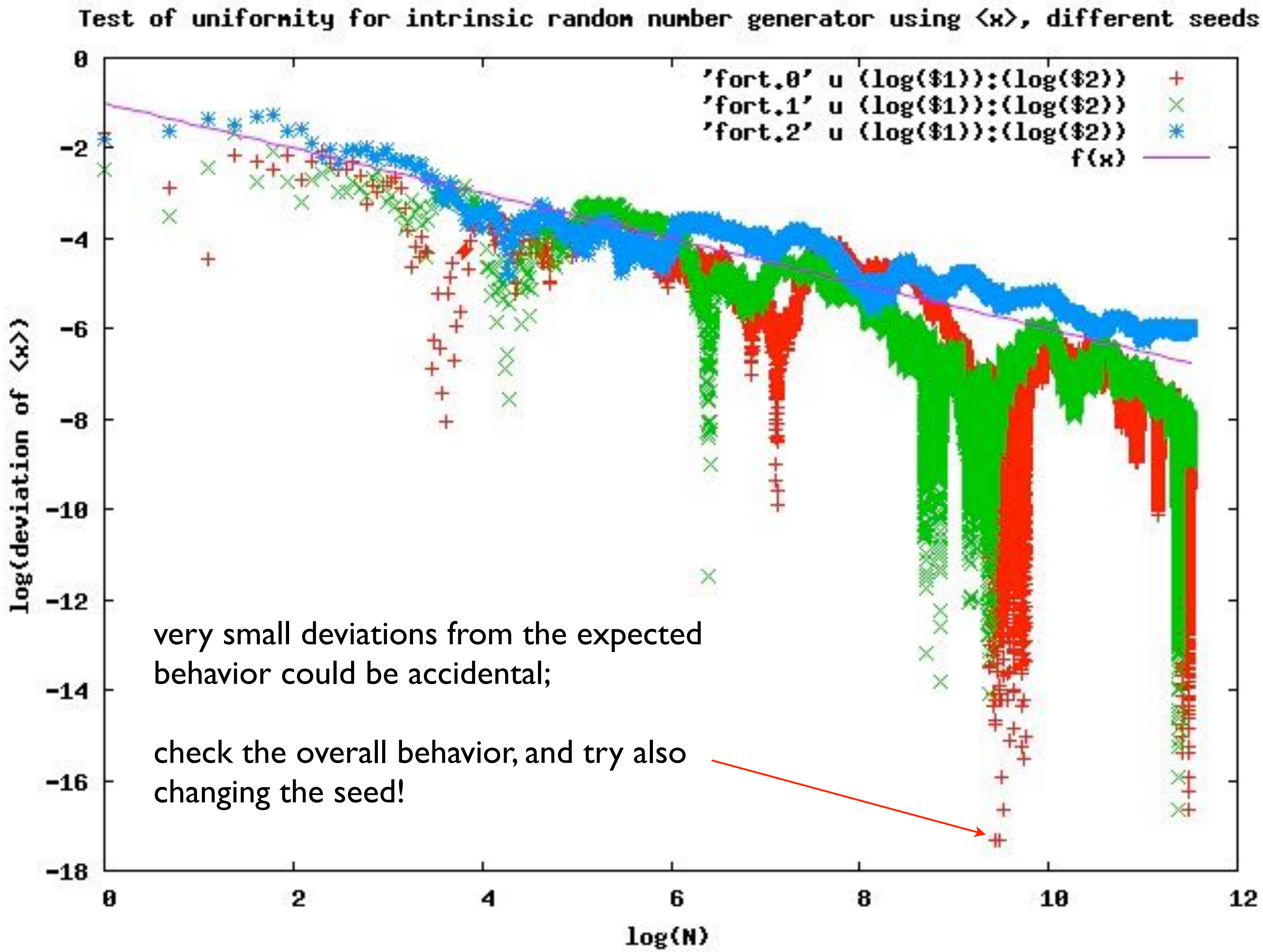
Test on one sequence, several momenta



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



A general suggestion:

do you want to check a power law?

deviation of <x>^{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

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 4:
use of the seed

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

allocate(seed(sizer))
```

Check how random_seed() works with gfortran or other compilers (g95...)
Do you want to force the seed initialization but not "by hands"?

Exercise 5 (optional):
how to change the seed using the computer clock

```
SUBROUTINE init_random_seed
INTEGER :: i, nseed, clock
INTEGER, DIMENSION(:), ALLOCATABLE :: seed

CALL RANDOM_SEED(size = nseed)
ALLOCATE(seed(nseed))
CALL SYSTEM_CLOCK(clock)

seed = clock/2 + 37 * (/ (i - 1, i = 1, n) /)
CALL RANDOM_SEED(PUT = seed)

DEALLOCATE(seed)
END SUBROUTINE
```

Exercise 6 - optional

nrdemo_ran.f90

```
module ran_module
   implicit none
   public :: ran_func
    contains

   FUNCTION ran_func(idum) result(ran)
         ...
   ...
   END FUNCTION ran_func

end module ran_module
```

```
   program demo
      use ran_module
      implicit none
      integer :: i,idum
      real :: x
      print*, "idum (<0) = "
      read*,idum
      x =ran_func(idum)
   ...
   end program demo
```

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
```