Generation of random numbers

• In many problems in scientific computing large numbers of **random numbers** (**RN**s) are needed.
  - Different simulation methods; particularly Monte Carlo[1].

• Different kinds of RNs:

1. **Real RN**s obtained from some physical processes (radioactive decay, electrical noise, ...)
   Impractical. (However, see www.random.org!)

2. **Pseudo RN**s are generated by computer by a deterministic algorithm but have the properties that are needed in applications.

3. **Quasi RN**s progressively cover a d dimensional space with evenly distributed points.
   These may sometimes give faster convergence for e.g. MC integration.

• In this chapter we talk about pseudo RNs (and drop the word 'pseudo').

• For a more thorough discussion on RNs attend the course 'Basics of Monte Carlo Simulations'
  (http://beam.acclab.helsinki.fi/~eholmstr/mc/)

• The starting point is a random number *generator* (**RNG**) that produces evenly distributed RNs in the interval $[0, 1)$.

1. In fact, the MC method in general could be defined as a method that uses large numbers of random numbers.

Generation of random numbers

• What properties do we expect from a decent RNG?

- **Statistical properties.** Even distribution. No correlations, not even for n-tuples of RNs.

- **Long period.** RNGs are created by computer subroutines $\rightarrow$ only finite number of them.

- **RN sequences should repeatable.** Needed when testing the application program:
  runs with different parameter values but with the exactly same RN sequence.
  Also, it should be possible to save the state of the RNG so that the run can be later continued.

- **Portability.** RNG should give exactly the same sequence on different platforms,
  independent on the word length and such.

- **Speed.** Maybe not so important nowadays; depends of course on the application.

- **Parallel computations.** RNG should parallellizable i.e. able to generate many
  non-overlapping sequences.

• Practically all RNGs are implemented by using integer arithmetics.

- By scaling the RN is converted to a floating point number within the interval $[0, 1)$.

Generation of random numbers: uniform RNs

• The oldest type of RNGs is the linear congruential generator (**LCG**).

- The RN sequence $\{x_i\}$ is computed as

$$x_{i+1} = (ax_i + c) \bmod m$$

- This generates RNs between 0 and $m-1$ $(c>0)$ or between 1 and $m-1$ $(c = 0)$.

- Parameters of the RNG are a, c and m, and it is often denoted as $\mathrm{LCG}(a, c, m)$.

- Parameter m is normally a large integer and it determines the period P of the RNG: $P \le m$

- The result is scaled to obtain a RN in the interval $[0, 1)$: $r = \frac{x_i}{m}$

- Constant c is sometimes dropped; then the RNG is called $\mathrm{MLCG}(a, m)$ (M=multiplicative).

- To start the sequence we need the initial seed x_0.

  - It can be given as an input from the user or it can be generated from the system state.
  - Both are needed:
    You change something in your program and want to see its effect →
    run with the exactly same RN sequence i.e. use the same seed.
    A huge number of runs to get statistics → they all need a unique seed.

Generation of random numbers: uniform RNs

- One way to get the seed is to compute it from the time of day.

C

```
NAME
 gettimeofday, settimeofday - get / set time

SYNOPSIS
 #include <sys/time.h>
 int gettimeofday(struct timeval *tv,
       struct timezone *tz);

DESCRIPTION
The  gettimeofday can get the time as well as a
timezone.   The tv argument is a timeval struct,
as specified  in
   <sys/time.h>:
   struct timeval {
     time_t      tv_sec; /* seconds */
     suseconds_t tv_usec; /* microseconds */
   };

and gives the number of seconds and  microseconds
since  the  Epoch (see time(2)).
```

Fortran

```
Syntax: CALL DATE_AND_TIME ([date]
         [, time] [, zone] [, values])

VALUES(1) is the 4-digit year.
VALUES(2) is the month of the year.
VALUES(3) is the day of the month.
VALUES(4) is the time difference with respect
             to Coordinated Universal Time
             (UTC) in minutes.
VALUES(5) is the hour of the day
             (range 0 to 23).
VALUES(6) is the minutes of the hour
             (range 0 to 59)
VALUES(7) is the seconds of the minute
             (range 0 to 59).
VALUES(8) is the milliseconds of the
             second (range 0 to 999)
```

Generation of random numbers: uniform RNs

- Below are examples of how to do it in C and Fortran

```
#include <stdlib.h>
#include <sys/time.h>
int getseed()
{
  int i;
  struct timeval tp;
  if (gettimeofday(&tp,(struct timezone *)NULL)
                          ==0) {
    i=tp.tv_sec+tp.tv_usec;
    i=(i%1000000)|1;
    return i;
  } else {
    return -1;
  }
}
```

```
function getseed()
  implicit none
  integer :: getseed
  integer :: t(8),i

  call date_and_time(values=t)
  getseed=t(7)+60*(t(6)+60*(t(5)+24*(t(3)-
           1+31*(t(2)-1+12*t(1)))))+t(8)
  seed=ior(iseed,1)

end function getseed
```


Generation of random numbers: uniform RNs

- In Linux (and in many other Unix systems) there is a device called **`/dev/urandom`**. It provides the user with a stream of random bytes.
- Below are routines that use the device to obtain the seed.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <fcntl.h>

#define MASK 2147483647

int getseed()
{
  int i,rdev,rnum,nb=sizeof(int),rs;
  struct timeval tp;

  rdev=open("/dev/urandom",O_RDONLY);

  if (rdev==-1) {
    if (gettimeofday(&tp,(struct timezone *)NULL)==0) {
      i=tp.tv_sec+tp.tv_usec;
      i=(i%1000000)|1;
      return i;
    } else {
      return -1;
    }
  } else {
    rs=read(rdev,&rnum,nb);
    if (rs>0) {
      return MASK&rnum;
    } else {
      return -1;
    }
  }

}
```

```
function getseed()
  implicit none
  integer :: getseed
  integer :: t(8),rn,is
  integer,parameter :: LMASK=2147483647
  integer,parameter :: LUN=676767
  character (len=80) :: rdev='/dev/urandom'
  logical :: openok,readok

  openok=.true.
  readok=.true.

  open(LUN,file=rdev,form='unformatted', &
     & access='stream',action='read',iostat=is)
  if (is/=0) then
     openok=.false.
  else
     read(LUN,iostat=is) rn
     if (is/=0) then
        readok=.false.
     end if
  end if
  if (openok) close(LUN)

  if (openok.and.readok) then
     rn=iand(rn,LMASK)
  else
     call date_and_time(values=t)
     rn=t(7)+60*(t(6)+60*(t(5)+24*(t(3)-1+ &
         & 31*(t(2)-1+12*t(1)))))+t(8)
  end if
  getseed=rn
  return
```

Generation of random numbers: uniform RNs

- Note that

  - There is also a device called **`/dev/random`**. It is guaranteed to give good quality (high-entropy) RNs. However, the read may block for a long time when system collects enough entropy from e.g. mouse movement, network traffic, etc.

  - The stream access of Fortran file io is a feature of Fortran 2003.
    Gnu Fortran (`gfortran`) supports it, as of version 4.2.
    Intel Fortran at least version 11.1 supports it.

Generation of random numbers: uniform RNs

- An below is the implementation of $\mathrm{LCG}(69069, 1, 2^{32})$ in C:

```c
double lcg(long long int *seed)
{
  static long long int a=69069,c=1,
                       m=4294967296; /* 2^32 */
  static double rm=4294967296.0;

  *seed=(*seed * a+c)%m;
  return (double)*seed/rm;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int getseed();
double lcg(long long int *iseed);

int main (int argc, char **argv)
{
  long long int seed;
  int i,imax;
  double x,y;

  imax=atoi(*++argv);
  seed=atoi(*++argv);
  if (seed<=0) seed=getseed();
  fprintf(stderr,"Seed %lld\n",seed);

  for (i=0;i<imax;i++) {
    x=lcg(&seed);
    y=lcg(&seed);
    fprintf(stdout,"%10f %10f\n",x,y);
  }
```

- Note the use of C **`long long int`** data type.

Generation of random numbers: uniform RNs

- An below is the implementation of $\mathrm{LCG}(69069, 1, 2^{32})$ in Fortran:

```
function lcg(seed)
  implicit none
  integer,parameter :: ik=selected_int_kind(15),&
                     & rk=selected_real_kind(10,40)
  real(rk) :: lcg
  integer(ik),intent(inout) :: seed
  integer(ik),save :: a=69069,c=1,m=4294967296
  real(rk),save :: rm=4294967296.0

  seed=mod(seed*a+c,m)
  lcg=seed/rm

end function lcg
```

```
program lcgmain
  implicit none
  integer,parameter :: ik=selected_int_kind(15),&
                     & rk=selected_real_kind(10,40)
  integer :: i,imax,iargc
  real(rk) :: x,y
  integer(ik) :: seed
  character(len=80) :: argu
  real(rk) :: lcg
  integer :: getseed

  call getarg(1,argu); read(argu,*) imax
  call getarg(2,argu); read(argu,*) seed
  if (seed<=0) seed=getseed()
  write(0,'(a,i10)') 'Seed ',seed

  do i=1,imax
     x=lcg(seed)
     y=lcg(seed)
     write(6,*) x,y
  end do

end program lcgmain
```

- Note the use of long integer data type.

Generation of random numbers: uniform RNs

- When compiling the C version with **gcc** in 32 bit Linux environment you have to tell the compiler to conform to the C99 standard:

```
lcg> gcc  lcgmain.c lcg.c getseed.c
lcg.c: In function 'lcg':
lcg.c:4: warning: integer constant is too large for "long" type
lcg> gcc -std=c99 lcgmain.c lcg.c getseed.c
lcg> a.out 10 4566
Seed 4566
  0.073428   0.569527
  0.691574   0.319013
  0.941832   0.372612
  0.971644   0.448007
  0.414919   0.060601
  0.674573   0.103366
  0.398162   0.648902
  0.034683   0.495077
  0.445557   0.142590
  0.526216   0.239509
lcg>
```

- Both versions were tested on 32 bit Linux Intel (`gcc`, `icc`, `ifort`), 64 bit Linux AMD Opteron (`gcc`, `pgcc`, `pgf90`), and on 64 bit HP Tru 64 Unix (`gcc`, `cc`, `f90`) environments.

  → All runs gave the same sequences.

- Making a RNG portable is not always this easy.

Generation of random numbers: uniform RNs

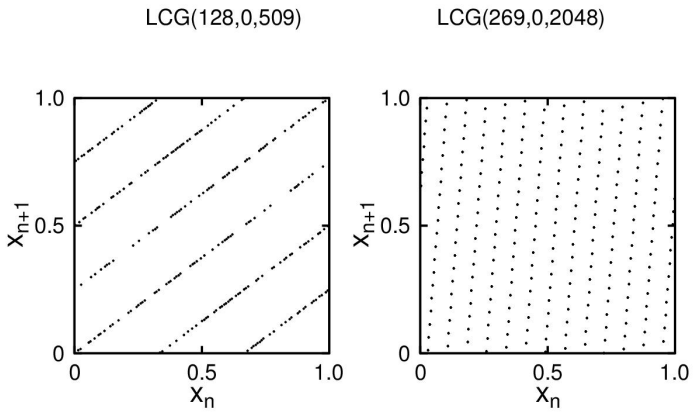
- In most cases in the LCGs found in the literature the values of the constants a, c, and m have been chosen carefully.
  - Don't go changing them if you don't know what you are doing.

- The state of a LCG RNG is described by only one number → period is limited to the range of that number.

- Probably the worst feature of the LCG RNGs is the correlations.

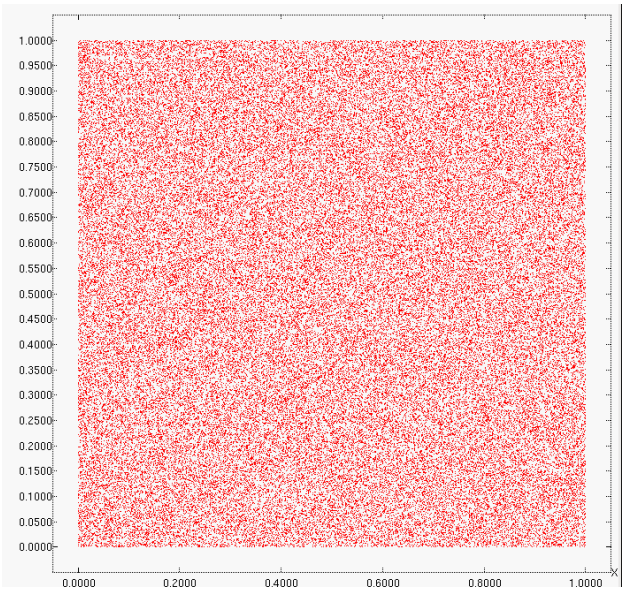
  - d-tuples of RNs $\{x_{i+1}, x_{i+2}, \ldots, x_{i+d}\}$ generated by a LCG form parallel hyperplanes in d dimensional space.

    - Hyperplanes are equidistant and the theoretical maximum number of them in a unit hypercube $[0, 1)^d$ is $(d!m)^{1/d}$.
    - The smaller the number of planes the worse is the distribution of the RNs in the cube.
    - A good LCG is one with parameters a and m that maximize the number of planes.
    - Below are examples of poor LCGs in 2D

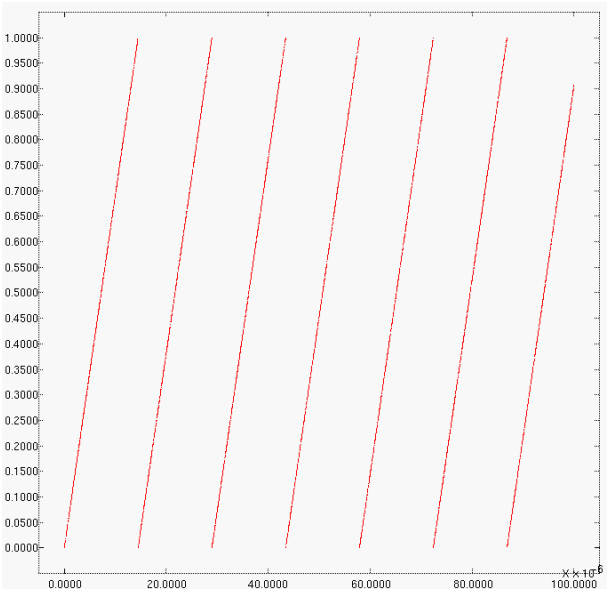


Generation of random numbers: uniform RNs

- These planes can be found also in the better LCGs when you know where to look for. The LCG from the examples above (note the x axis scale in the right hand side figure):



$x, y \in [0, 1]$



$x \in [0, 10^{-4}]$, $y \in [0, 1]$

- Command lines:

```
lcg> ./lcgmain 100000 987345 | xgraph -p -tk -nl
lcg> ./lcgmain 100000000 987345 | xgraph -p -tk -nl
```

```
if (x<1e-4) printf("%20.15g %20.15g\n",x,y);
```

Generation of random numbers: uniform RNs

- Because we have to multiply two 32-bit integers the intermediate result may not fit to 32 bits.

- Another workaround (instead of using 64 bit integers) is Schrage's algorithm for multiplying two 32-bit integers modulo a 32-bit constant, without using any intermediates larger than 32 bits (including a sign bit).

  - It is based on the approximate factorization of integer m:

$$m = aq + r, \quad q = \left[\frac{m}{a}\right], \quad r = m \bmod a \quad ([x] \text{ means: integer part of})$$

$$\rightarrow \quad (ax) \bmod m = a(x \bmod q) - r\left[\frac{x}{q}\right]$$

  - For example the RNGs provided for exercise 9 are LCGs using $a = 16807$, $m = 2^{31} - 1 = 2147483647$ and Schrage's algorithm with $q = 127773$ and $r = 2836$[1]:

```
double myrand(int *seed) {
  static int   a=16807, m=2147483647, q=127773, r=2836;
  double minv = (double) 1.0/m;

  *seed = a*(*seed % q)-r*(*seed / q);
  if (*seed < 0) *seed = *seed + m;
  return (double) *seed * minv;
}
```

1. Minimal RNG of Park and Miller. This the `ran0` RNG of Numerical Recipes.

Generation of random numbers: uniform RNs

• Lagged Fibonacci generators (**LFG**) are based on the generalization of the LCG.

- The period of the LCG can be increased by the form

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \ldots + a_p x_{i-p}) \bmod m,$$

where $p > 1$ and $a_p \neq 0$.

- A LFG requires an initial set of elements $x_1, x_2, \ldots, x_r$ and then uses integer recursion

$$x_i = (x_{i-q} \otimes x_{i-r}) \bmod m,$$

where q and r are integer *lags* satisfying $q > r$ and $\otimes$ is one of the following binary operations:

+	addition
–	subtraction
×	multiplication
⊕	exclusive-or

Generalization of the Fibonacci sequence $x_i = x_{i-1} + x_{i-2}$

- The corresponding generators are termed $\mathrm{LFG}(r, s, \square)$.

- Initialization requires q random numbers which can be generated by e.g. a LCG.

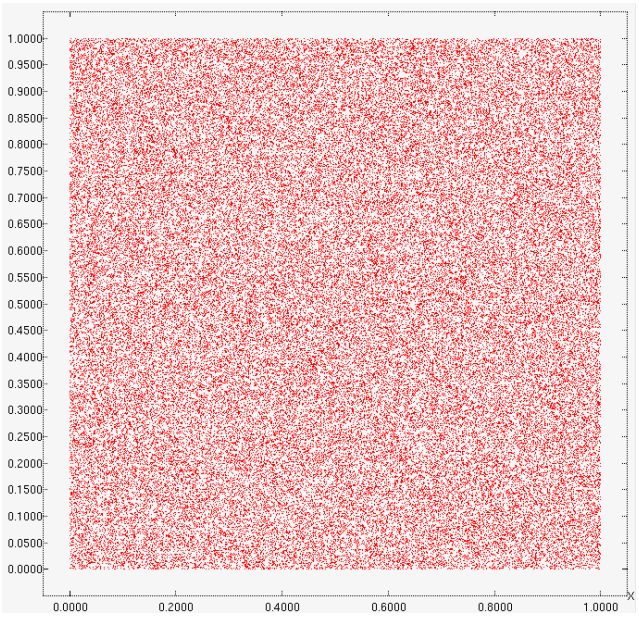
Generation of random numbers: uniform RNs

- An example of LFG is the $\mathrm{LFG}(55, 24, -)$ (e.g. the **`ran3`** routine of Numerical Recipes):

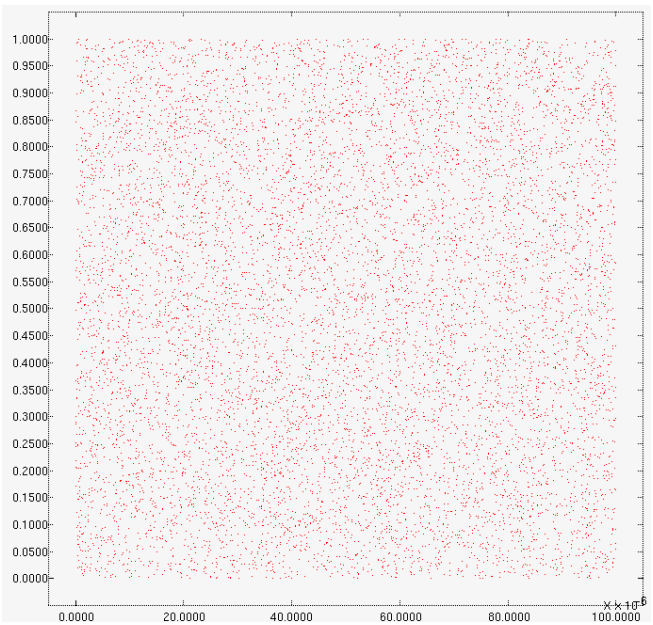
```
#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)

double lfg(long int *seed)
{
  static int inext,inextp;
  static long int ma[56];
  static int first=1;
  long mj,mk;
  int i,ii,k;

  /* Initialization */
  if (*seed < 0 || first) {
    first=0;
    mj=MSEED-(*seed < 0 ? -*seed : *seed);
    mj %= MBIG;
    ma[55]=mj;
    mk=1;
    for (i=1;i<=54;i++) {
       ii=(21*i) % 55;
       ma[ii]=mk;
       mk=mj-mk;
       if (mk < MZ) mk += MBIG;
       mj=ma[ii];
     }
     for (k=1;k<=4;k++)
       for (i=1;i<=55;i++) {
  ma[i] -= ma[1+(i+30) % 55];
  if (ma[i] < MZ) ma[i] += MBIG;
       }
    inext=0;
    inextp=31;
    *seed=1;
  }

  /* Generation */
  if (++inext == 56) inext=1;
  if (++inextp == 56) inextp=1;
  mj=ma[inext]-ma[inextp];
  if (mj < MZ) mj += MBIG;
  ma[inext]=mj;
  return mj*FAC;
}
```


Generation of random numbers: uniform RNs

- Similar runs as for the LCG:



$x, y \in [0, 1]$



$x \in [0, 10^{-4}]$, $y \in [0, 1]$

- No d-tuple correlations in this LFG.

- However, in some MC simulations correlations have been observed.
  - Remedy: take only every second or third number in the sequence (decimation strategy).

Generation of random numbers: uniform RNs

• Shift-register generators (**SRG**s) can be viewed as the special case $m = 2$ of LFGs.

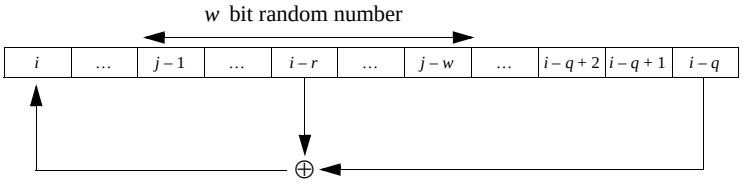
- They are based on bit sequences $b = \{b_i\}$ with recursive definition

$$b_i = (c_1 b_{i-1} + c_2 b_{i-2} + \ldots + c_w b_{i-w}) \bmod 2$$

- Coefficients c_j $(j = 1, \ldots, w-1)$ are either one or zero and $c_w = 1$, which guarantees that the w-tuples of the bit sequence $\{b_{i-1}, b_{i-2}, \ldots, b_{i-w}\}$ has the maximum period of $2^w - 1$.

- In practice coefficients c_j are chosen according to equation $x_i = (x_{i-q} \otimes x_{i-r}) \bmod m$ so that $c_q, c_r \neq 0$ and others are zero.

- By choosing the operation to be exclusive or and setting $m = 2$ we get the following recursion



- Problem with this RNG is the short period.

Generation of random numbers: uniform RNs

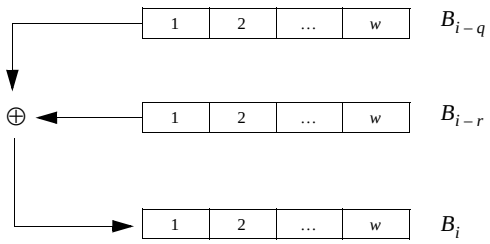
- In the generalized feedback shift-register generators (**GFSRG**s) bit-sequences $B_i = \{b_{j+1}, b_{j+2}, \ldots, b_{j+w}\}$ corresponding to integers are handled.

- The binary operation $\oplus$ is applied to two bit sequences instead of single bits:

$$B_i = (c_1 B_{i-1} + c_2 B_{i-2} + \ldots + c_q B_{q-1}) \bmod 2$$

- Coefficients c_j are usually chosen according the equation $x_i = (x_{i-q} \otimes x_{i-r}) \bmod m$.

- Schematically the recursion looks like



- The good point in the GFSRGs is the long period.

- However, correlations have been observed in some test.
- They can be avoided by the decimation strategy: use only every second or third RN.

Generation of random numbers: uniform RNs

- The best choices for r and q are Mersenne primes (if r is a prime $2^r - 1$ is also a prime) and for which

$$r^2 + q^2 + 1 = \text{prime}$$

- Examples of pairs satisfying this are

$$r = 98 \quad q = 27$$
$$r = 250 \quad q = 103$$
$$r = 1279 \quad q = 216, 418$$
$$r = 9689 \quad q = 84, 471, 1836, 2444, 4187$$

- GFSRGs are denoted by $\mathrm{GFSR}(r, q, \Uparrow)$

Generation of random numbers: uniform RNs

- One implementation of GFSRGs is the **R250** RNG where $r = 250$, $q = 103$.

```
#define NR 1000
#define NR250 1250
#define NRp1 1001
#define NR250p1 1251

double lcgy(int *seed) {
  static int   a=16807, m=2147483647,
              q=127773, r=2836;
  double minv = (double) 1.0/m;
  *seed = a*(*seed % q)-r*(*seed / q);
  if (*seed < 0) *seed = *seed + m;
  return (double) *seed * minv;
}

void r250(int *x,double *r,int n)
{
  static int q=103,p=250;
  static double rmaxin=2147483648.0; /* 2**31 */
  int i,k;
  for (k=1;k<=n;k++) {
    x[k+p]=x[k+p-q]^x[k];
    r[k]=(double)x[k+p]/rmaxin;
  }
  for (i=1;i<=p;i++) x[i]=x[n+i];
}
```

```
double ran_number(int *seed)
{
  double ret;
  static int firsttime=1;
  static int i,j=NR;
  static int x[NR250p1];
  static double r[NRp1];
  if (j>=NR) {
    if (firsttime==1) {
      for (i=1;i<=250;i++)
        x[i]=2147483647.0*lcgy(seed);
      firsttime=0;
    }
    r250(x,r,NR);
    j=0;
  }
  j++;
  return r[j];
}
```

Generation of random numbers: uniform RNs

- And the main program using the RNG:

```
#include <stdio.h>

int getseed();
double ran_number(int *seed);

int main(int argc, char **argv)
{
  int seed;
  int i,imax;
  double x,y;

  imax=atoi(*++argv);
  seed=atoi(*++argv);
  if (seed<=0) seed=getseed();
  fprintf(stderr,"Seed %d\n",seed);

  for(i=0; i<imax; i++) {
    x=ran_number(&seed); y=ran_number(&seed);
    fprintf(stdout,"%g %g\n",x,y);
  }
  return(0);
}
```

- R250 does not have the d-tuple correlations but fails in some physical tests like random walk and MC simulation of the Ising model.
- As already said by taking only every k^{th} RN ($k = 3, 5, 7, \ldots$) reduces correlations.

Generation of random numbers: uniform RNs

• RNGs can be tested by various ways.

- Theoretical tests based on number theoretical arguments
- Empirical test: compute some statistics of the RN sequence
  - Distribution: should be even !
  - χ^2 test $\chi^2 = \sum_{i=1}^{N} \left(\frac{y_i - y(x_i)}{\sigma_i} \right)^2$
  - Spectral test: distribution of n- tuples $\{x_i, x_{i+1}, \ldots, x_{i+n}\}$ of RNs in n-dimensional space (remember hyperplanes in LCG RNGs)
  - Testing in real application (MC simulations)
  - Visual inspection: human eye good in extracting patterns:

Mertens & Bauke, *Phys. Rev. E*, **69** (2004) 055702.

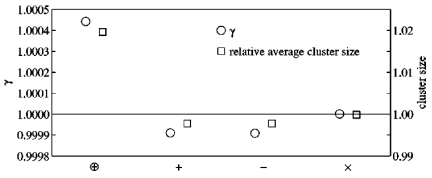
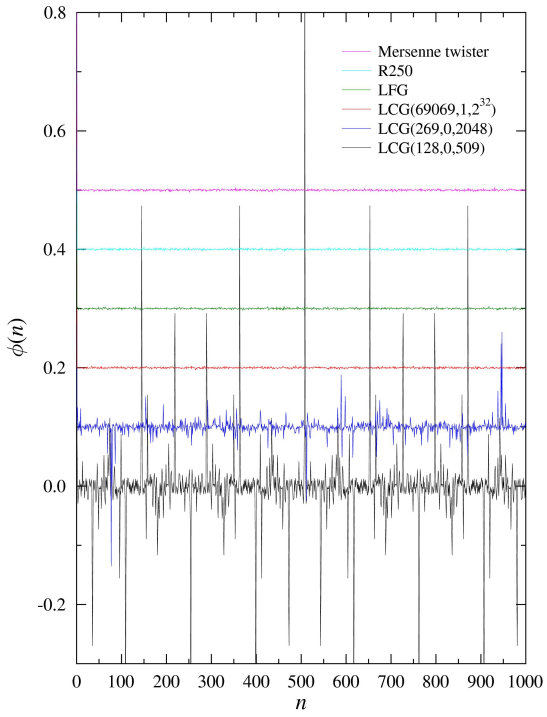
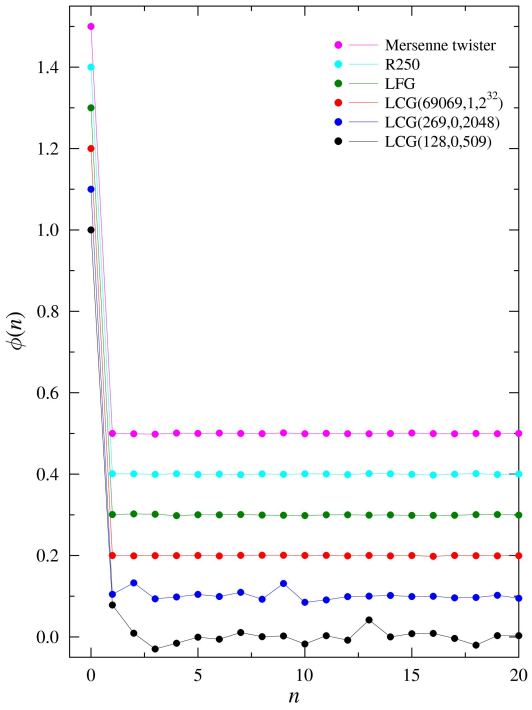


FIG. 2. Bias in the output of the lagged Fibonacci generator with biased inputs (○) and cluster sizes in the Wolff algorithm (□). The bias in the input numbers was $\gamma = 0.975$ and $P_{\text{add}} = 0.586$, corresponding to a MC simulation at the critical temperature of the 2D Ising model. The simulation was done with $F(13, 33, \circ)$ on a 16 × 16 spin system. Error bars are smaller than the symbol size. The reference value of the cluster size has been obtained from simulations with a high quality PRNG [18].

Generation of random numbers: uniform RNs

- Below are plotted the autocorrelation functions of various RNGs $\phi(n) = \dfrac{\langle r_1 r_n \rangle - \langle r \rangle^2}{\sigma(r)}$.





Generation of random numbers: uniform RNs

- Below is shown (just for curiosity) the standard deviation of a distribution $d(r)$ of uniform random numbers r in the interval $[0, N_d - 1)$ (generated by the Mersenne twister).

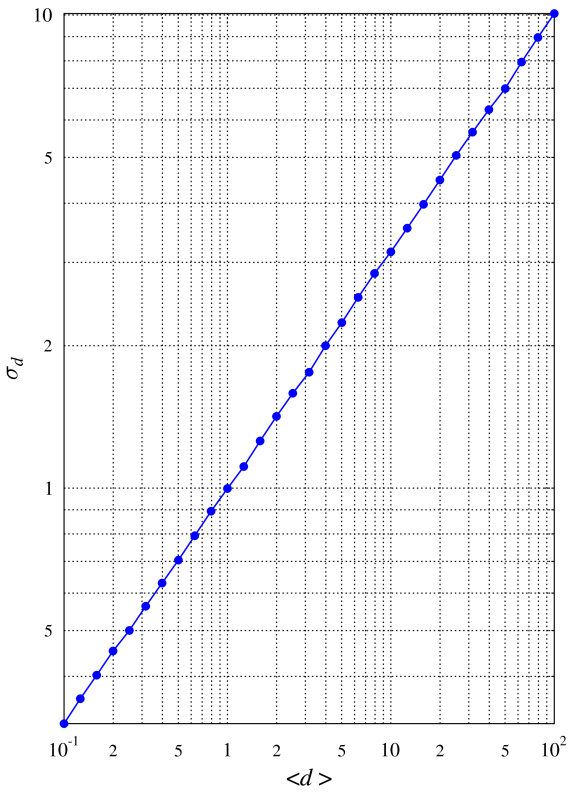
$$\sigma_d = \sqrt{\langle d^2 \rangle - \langle d \rangle^2}$$

$$\langle d \rangle = \frac{1}{N_d} \sum_{i=0}^{N_d} d_i$$

$$\langle d^2 \rangle = \frac{1}{N_d} \sum_{i=0}^{N_d} d_i^2 .$$

- It is easy to see that

$$\sigma_d = \sqrt{\langle d \rangle} .$$



Generation of random numbers: uniform RNs

• As we have seen all the RNGs presented have some sort of correlations.

- One way to reduce these correlations is to combine two RNGs:

$$z_i = x_i \otimes y_i ,$$

where x_i, y_i are RN from some (good) RNGs and $\otimes$ is a binary operator (+,-,×,⊕).

- One example of combination RNGs is RANMAR which combines LFG with simple arithmetic sequence.

- RANMAR is rather well tested and has the period about 2^{144}.

- Implementation: http://www.physics.helsinki.fi/courses/s/tl3/progs/rng/ranmar/

• Gnu Scientific Library (GSL) has many high-quality RNGs implemented. Check out the info-pages.

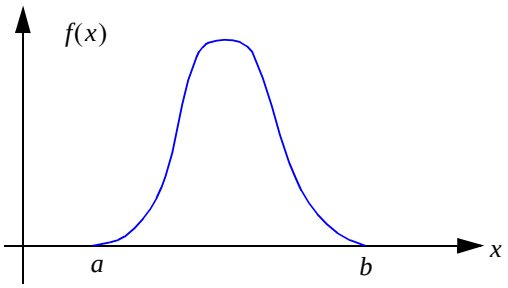
• One of the state-of-the-art RNGs is the **Mersenne Twister**.
- It belongs to the generalized feedback shift register class.
- A Fortran implementation can be found in course web page at
http://www.physics.helsinki.fi/courses/s/tl3/progs/rng/mersennetwister/
- For other implementations check out the RNG home page at
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html

Generation of random numbers: non-uniform RNs

• From the uniform RNs we need to get RNs that have various distribution.

- Let the desired distribution be $f(x)$ in the interval $[a, b]$:

$$f(x) \geq 0, \forall x \in [a, b]$$
$$\int_a^b f(x)dx = 1 .$$



- Let $F(x)$ be the cumulative distribution

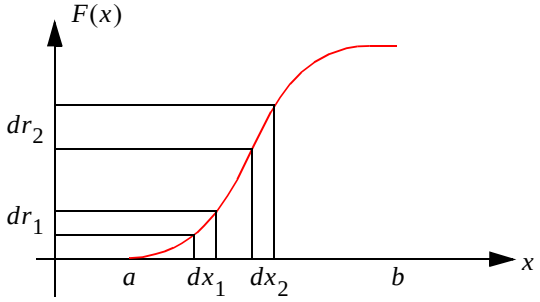
$$F(x) = \int_a^x f(x')dx' \equiv r .$$

- According to definition we can map $F(x)$ to a random variable $r \in [0, 1]$.

- Let's investigate the two equal subintervals dx_1 and dx_2 at positions x_1 and x_2, respectively.

- We can easily see that

$$\frac{dr_1}{dr_2} = \frac{(dF(x)/dx)|_{x=x_1}}{(dF(x)/dx)|_{x=x_2}} = \frac{f(x_1)}{f(x_2)}.$$



Generation of random numbers: non-uniform RNs

- I.e. by generating uniformly distributed RNs the ratio of hits in region dr_1 to that in dr_2 is the ratio of the probability densities at these points.

- Thus, we can get RNs with the desired distribution $f(x)$ by doing the transformation

$$x = F^{-1}(r).$$

- Note that all functions $F(r)$ that are cumulative probability distributions have an inverse function.

- As an example we'll take the free path of a photon in matter.

  - The probability density distribution (in units of mean free path) is

$$f(z) = e^{-z}.$$

  - Cumulative distribution is

$$F(z) = \int_0^z e^{-z'} dz' = 1 - e^{-z}$$

  - Inverting this we get

$$z = -\ln(1-r)$$

  - Because $1-r$ and r have similar distributions we get the final answer

$$z = -\ln r.$$

Generation of random numbers: non-uniform RNs

• It is not always possible to form the inverse function of F (although it does exist).

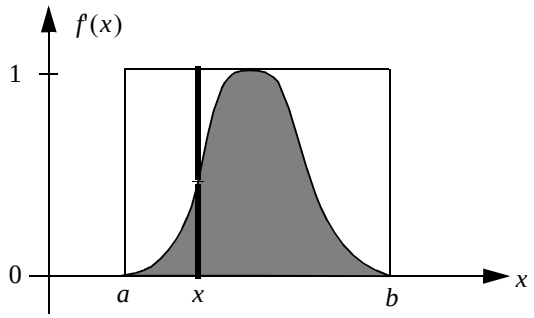
- In these cases we can use the rejection method.

1. Form a new function from the probability distribution

$$f'(x) = \frac{f(x)}{f_{\max}}$$

2. Generate a uniform RN $r_1 \in [0,1]$ and normalize this to interval $[a,b]$: $x = a + (b-a)r_1$.
   If $[a,b]$ is not finite one has to transform it into one.
   E.g. $x \in [a,\infty]$ can be transformed as $x = a[1-\ln(1-y)]$ to interval $y \in [0,1]$.

3. Generate a random number $r_2 \in [0,1]$. If $r_2 < f'(x)$, accept x otherwise go to step 2.



- The drawback of the method is that if $f(x)$ is peaked we waste many RNs in rejections.

Generation of random numbers: non-uniform RNs

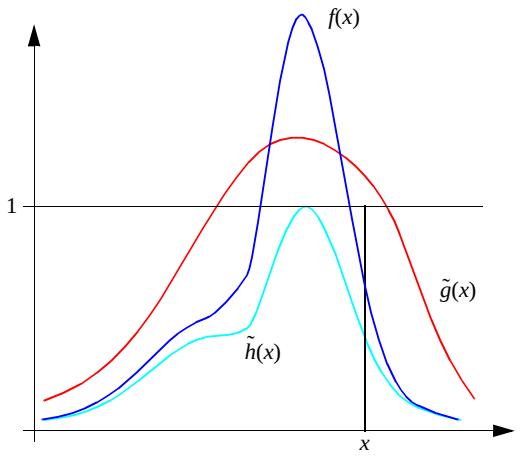
• We can also use a combination of the two methods presented above.

- First separate the probability distribution into two parts

$$f(x) = g(x)h(x) .$$

where $g(x)$ is easily inverted and contains most of the peakedness
and $h(x)$ is rather smooth but mathematically complicated so that it can not be inverted.

- Now RNs can be generated as below:



1. Scale $g(x) \to \tilde{g}(x)$ so that $\int_a^b \tilde{g}(x)dx = 1$.

2. Scale $h(x) \to \tilde{h}(x)$ so that $\tilde{h}(x) \le 1 \quad \forall x \in [a, b]$.

3. Using the inversion method generate RNs with distribution $\tilde{g}(x)$.

4. Using RN x apply the rejection method with distribution $\tilde{h}(x)$:
   Generate a RN $r \in [0, 1]$. If $\tilde{h}(x) > r$ accept x otherwise
   go to step 3.