

3. Monte Carlo - introduction

3.1. Definition

http://www-groups.dcs.st-andrews.ac.uk/history/HistTopics/Pi_through_the_ages.html

In the widest sense of the term, **Monte Carlo** (MC) simulations mean any simulation (not even necessarily a *computer* simulation) which utilizes random numbers in the simulation algorithm.

The term “Monte Carlo” does indeed come from the famous casinos in Monte Carlo. When study of MC methods became in earnest in the 1940’s and 1950’s, someone thought of the likeness of using random numbers to the randomness in the games at the casino, and named the method thereafter.

Another closely related term is **stochastic** simulations, which means the same thing as Monte Carlo simulations. Even though stochastic would seem to be a more descriptive term, since the chances of misunderstandings are much smaller, Monte Carlo is more widely used probably for no better reason than that it sounds more exciting!

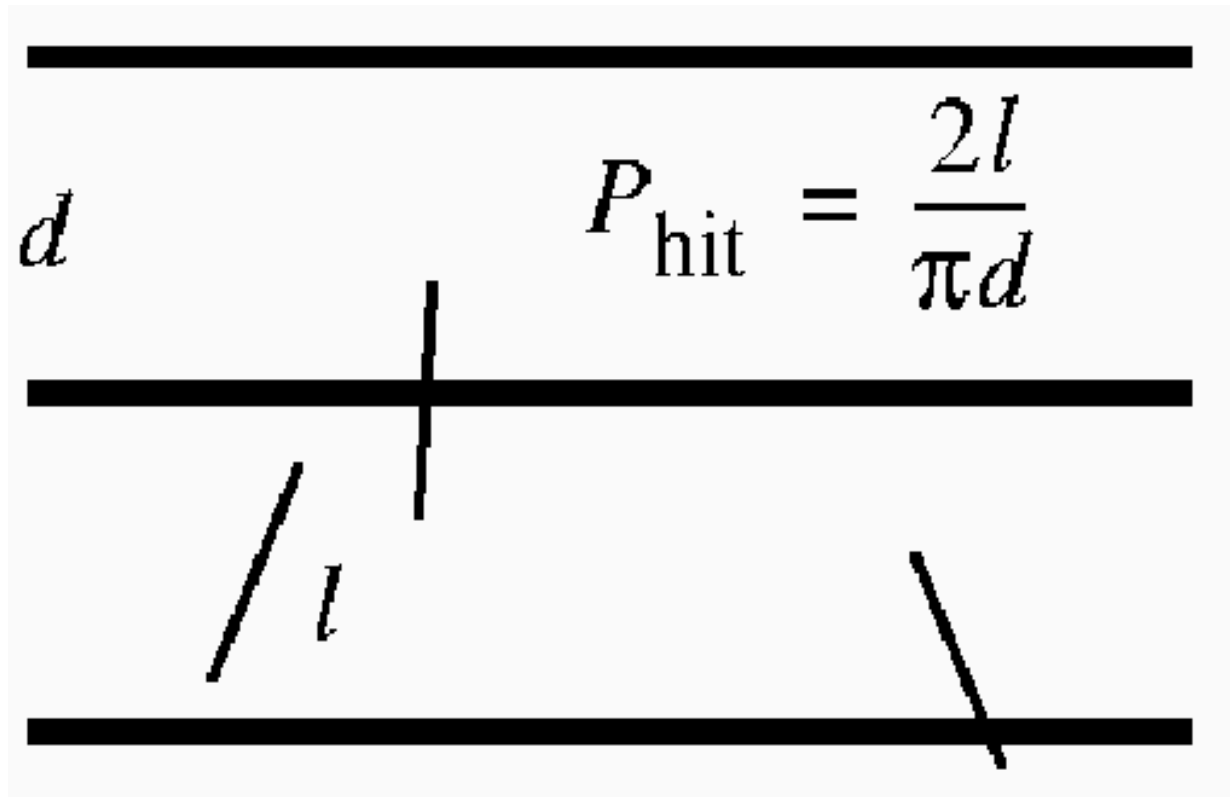
The idea of using randomness to obtain an answer to a scientific problem is actually much older than computers.

3.2. Very brief history

It was Buffon (1707 - 1788) who found out that if one has a uniform grid of parallel lines, distance d apart, and if one drops a needle of length $l < d$ on the grid, the probability that the needle falls across a line is

$$P_{hit} = \frac{2l}{\pi d}$$

(Proving this is left as an exercise).



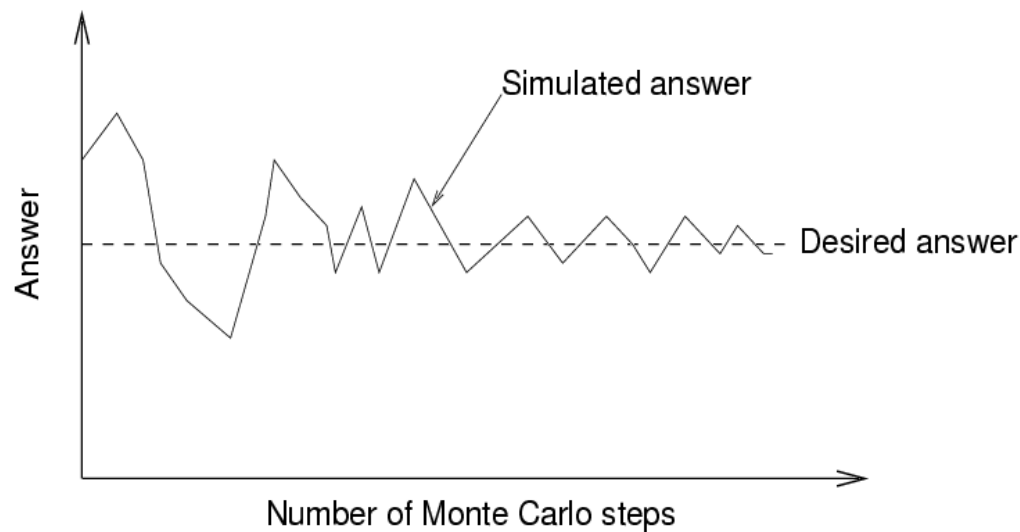
Various people have tried to calculate π by throwing needles. The most remarkable result was that

of Lazzarini (1901), who made 3408 tosses and got

$$\pi = \frac{355}{113} = 3.1415929$$

This outcome is suspiciously good.

This actually nicely illustrates a potential pitfall in MC simulations. If you know in advance the answer you want to get, and start doing MC to obtain it looking after every step at your intermediate answer, your result will behave something like follows:



So if you stop the simulation exactly when you happen to cross the 'right answer' line, you will get the right answer! But doing this is of course utterly wrong.

In practice, people seldom do MC simulations knowing the answer in advance (what would be the point of the simulation). But a more dangerous and common pitfall is that you e.g. wish for a low, or high, value, you stop the simulation at a value in one of the ends.

So pretty obviously, what Lazzarini did is stop his needle throwing when he got the accurate answer, which explains the strange number 3408 of tosses.

To avoid this pitfall, one should always decide in advance how many MC steps to do.

Since Buffon's needle, random numbers were used now and then in statistics, e.g. by Student (in connection with developing the distribution which carries his name). But the wide use began really in the 1940's (see links on web page). Exactly where and how it began and where the name originates from is hard to pin down exactly.

This is rather obviously is related to the fact that the beginnings were closely related to the Manhattan project, and publication of results was not the number one priority there....

If you check the web for this, be careful: some sources are in direct contradiction to others.

Two early crucial papers are:

Note on census-taking in Monte Carlo calculations E. Fermi and R.D. Richtmyer 1948. A declassified report by Enrico Fermi. From the Los Alamos Archive.

The Monte Carlo Method N. Metropolis and S. Ulam 1949 Journal of the American Statistical Association, 44, 335 (1949)

The name apparently originates with Metropolis, who says that he invented it in 1945 inspired by Stan Ulam's gambling uncle who "just had to go to Monte Carlo" [Metropolis, N. "The Beginning of the Monte Carlo Method." Los Alamos Science, No. 15, p. 125, <http://jackman.stanford.edu/mcmc/metropolis1.pdf>]

3.3. MC now, and intent of this course

Nowadays MC methods are also used widely outside of mathematics, statistics, and physics. There are uses in:

- Optimization (any field where it is needed)

- Social sciences: there is even a book called “Monte Carlo Simulation (Quantitative Applications in the Social Sciences)”

- Economics: citation from the description of the book “Monte Carlo Methods in Financial Engineering”:

“Monte Carlo Methods are among the most broadly applicable and thus most powerful tools for valuing derivatives securities and measuring their risks. ”

Common to all of the wide areas where MC is used is that one in any of them one has to be able

to generate random numbers efficiently. Many of the basic MC methods, such as MC integration and generation of synthetic data, are also used in a wide range of sciences.

The intent of this course “Basics of MC” is specifically to go through the basic MC tools which can be used in a wide range of sciences.

3.4. Terminology

The term “MC” is actually extremely wide, and there is much overlap.

To give a concrete example, the molecular dynamics simulations which describe atomistic motion are in principle completely deterministic in nature. But in practice, one often uses random numbers in MD as well. The initial atom velocities are usually given randomly, so this brings in an MC character. But only because of this, MD simulations are not called MC.

But in more advanced cases MD is often used within an MC scheme. As a concrete example, consider for instance a radiation physics problem: we have a particle which impacts on a lattice. (FIGURE DRAWN ON LECTURE). The outcome (how deep in and where does it go) will be strongly dependent on the impact position. Hence one needs to simulate numerous (tens – thousands) of events to get a comprehensive picture of what can happen, and collect statistics. The initial impact positions are chosen randomly, so the set of simulations forms an MC simulation. The whole scheme could be called MC-MD.

I list here some of the more common MC terms and briefly describe what they are.

Metropolis MC

A simulation algorithm, central to which is the formula which determines whether a process should happen or not. Originally used for simulating atom systems in an NVT thermodynamic ensemble, but nowadays generalized to many other problems:

Markov Chain MC, MCMC

A generalized version of Metropolis MC, used to generate a sequence of points in some known distribution.

Simulated annealing

The Metropolis MC idea generalized to optimization, i.e. finding minima or maxima in a system. This can be used in a very wide range of problems, many of which have nothing to do with physics.

Thermodynamic MC

MC when used to determine thermodynamic properties, usually of atomic systems.

Lattice MC, LMC

MC used on a lattice. In condensed matter physics this is used to distinguish MC done on crystal lattices from those done in a random/amorphous medium

Kinetic MC, KMC

MC used to simulate activated processes, i.e. processes which occur with an exponential probability

$$e^{-E_a/k_B T}$$

Typical example is the migration of defects in a solid. This can be done either in a non-directional system, or on a lattice. In the latter case one could also talk about lattice kinetic MC.

Variational MC, VMC

Diffusion MC, DMC

These terms, although very wide in principle, are nowadays used often to signify a variety of electronic structure calculations where MC techniques are used to obtain the ground state electron configuration. Sometimes the terms are used without specifying that one deals with electronic

structure, which may lead to confusion as “diffusion MC” obviously could mean many other things as well.

Quantum Monte Carlo, QMC

Another term which is used in many different contexts. One is that the electronic structure calculation methods VMC and DMC are often called QMC. Another is that quantum mechanical simulations of spin systems (which does not necessarily relate to electronic structure-determining calculations in any direct way) are also called QMC...

In general, these examples emphasize that one should avoid using any XXXMC term and abbreviation without (at least in the beginning of the text) clearly defining precisely which context it is related to.

4. Generating random numbers

Central to any MC simulation are the random numbers. Hence it is important to have a good source of random numbers available for the simulations. Getting 'good' random numbers is in fact not quite as easy as many people think it is, so we will spend quite some time on this topic.

Random numbers can be divided into (at least) three categories: **true random** numbers, **pseudorandom** numbers and **quasirandom** numbers.

- True random : no way to predict what the next random number is
- Pseudorandom: a sequence of numbers which is algorithmically produced and not really random
 - It can be fully repeated if the initial conditions and algorithm are known
- Quasirandom numbers: act as random numbers in some sort of simulations, but are well-ordered in some other types.

More precise explanations for these categories are given later in this section.

4.1. Non-automated means

Although these are almost never used anymore, it is good to keep in mind that there do exist non-automated means to generate random numbers.

The most obvious one is to just write one manually.

Although that has obvious problems for generating large sets of numbers, it is widely used for selecting the seed number (see below) for electronic random number generators.

Another one, which was historically used to some extent, is to select numbers from some number sequence, e.g. the phone book or the decimals of π .

- The former method is highly nonadvisable, as there obviously can be strong non-random features in phone numbers.
- The latter is not so bad, since the decimals of π are not supposed to have correlations

One could for instance get rough random numbers between 0 and 1 by selecting always 4 numbers at a time from π and divide these by 10000:

3.141592653589793238462643383279502884197169399375105

These non-automated random numbers are essentially pseudorandom.

4.2. Mechanically generated random numbers

[Knuth, Seminumerical algorithms]

As technology evolved, mechanical devices were devised which produced random numbers.

- The machines used in lottery or roulette
- Also specifically designed machines which could generate thousands of numbers have been built. In 1955 the RAND corporation actually published a book with 1 million random numbers. (undoubtedly a strong contender for the title of most boring book ever published).

Since the advent of electronic computers, such devices have become obsolete, except in applications (such as the lotto games) where it is important that even a layman can easily check that there is no scam involved.

Mechanical random numbers can be true random numbers, such as the ones generated in a lotto machine or a roulette, (unless maybe if the casino is run by the mob...). Numbers generated by a mechanical calculating device may also be pseudorandom.

4.3. Electronic hardware-generated random numbers

[<http://www.fourmilab.ch/hotbits/>]

4.3.1. True random numbers

It might seem an obvious idea to design microprocessors, or parts of them, to be able to generate random numbers electronically - that is, design an electronic part which delivers a signal which randomly gets translated to 0's and 1's when they are translated to digital form. But this is in practice very rarely done.

But even an electronic random number generator could have its problems; it is easy to imagine that minute electronic disturbances from the environment could affect the results produced by it.

For a physicist, an obvious idea to obtain random numbers independent of any reasonably possible outside disturbance is to use radioactive decay. Since nuclear processes are almost completely

independent of all every-day electronic interactions, this should (if properly calibrated) deliver true randomness.

A quick google search reveals that there does indeed exist random numbers based on radioactivity, see <http://www.fourmilab.ch/hotbits/>. Here is a sample of 16 truly random bytes:

```
unsigned char hotBits[16] = 168, 193, 241, 195, 37, 251, 190, 121, 105, 137, 173, 226, 95, 181, 239, 231 ;
```

But unfortunately downloading random numbers from the internet would be way too slow for most MC simulations, and putting a radioactive source inside a chip is not a very good idea either.

4.3.2. Pseudorandom numbers

Before we discuss the common algorithmic pseudorandom numbers, we mention something which is something between true and pseudorandom numbers.

4.3.2.1. HAVEGE – HArdware Volatile Entropy Gathering and Expansion

[<http://www.irisa.fr/caps/projects/hipsor/HAVEGE.html>]

There is a serious effort going on in generating random numbers based on computer hardware. This is not done in the chip electronics, but by software which reads chip registers.

- These are unpredictable in the sense that it is practically impossible afterwards to know what values they had at the moment of reading.

This is especially important for modern cryptography, which heavily relies on random numbers.

- In case cryptography uses a pseudorandom number generator where the seed could be somehow figured out, then knowledge of what algorithm is used could be used to help crack the code. Hence it is important to have absolutely non-repeatable random number generators.

There are several such approaches, but most are pretty slow, capable of producing only 10's or 100's of bits per second.

In a quite new (2002 on) development, Seznec and Sendrier have formed a generator which uses several advanced features of modern superscalar processors, such as caches, branch predictors, pipelines, instruction caches, etc. Every invocation of the operating system modifies thousands of these binary volatile states. So by reading from them, one can obtain data which will immediately change afterwards and hence is practically impossible to reproduce.

So for practical purposes they do generate true random numbers, although formally they do not.

“HAVEGE (HARdware Volatile Entropy Gathering and Expansion) is a user-level software unpredictable random number generator for general-purpose computers that exploits these modifications of the internal volatile hardware states as a source of uncertainty.”

“During an initialization phase, the hardware clock cycle counter of the processor is used to gather part of this entropy: tens of thousands of unpredictable bits can be gathered per operating system call in average.” In practice, this is done for 16384 entries in a storage array called “Walk”.

After this, HAVEGE works as a pseudorandom number generator which keeps modifying the Walk array. But additional hardware-generated uncertainty is read in and used to read entries from the Walk table in a chaotic order. Also, hardware-generated data is used to modify the Walk table entries. Their pseudorandom-number generator relies on XOR operations (see below). Moreover, the data is hidden so that even the user of the routine can not read it.

It is even possible to randomly personalize the generator source code during installing, and not only with numbers but actually even by randomizing bit-level logical and shift *operators* in the source code.

On a 1GHz Pentium, the routine can throughput randomness at a rate of 280 Mbit/s. On Pentium III's, the uncertainty is gathered from the processor instruction cache, data cache, L2 cache, as well as the data translation buffer.

This routine clearly seems very promising for efficient generation of random bits (numbers) on modern processors.

For scientific simulations, as discussed in the next section, it is actually usually desirable to use **predictable** random numbers, to enable repeatability. For this kind of application, HAVEGE is clearly unsuitable.

4.4. Algorithmic pseudorandom numbers

The by far most common way of obtaining random numbers is by using some algorithm which produces a seemingly random sequence of numbers. But the numbers are not truly random; if the call to the routine is repeated with the same input parameters, the numbers will always be the same. Such numbers are called **pseudorandom** numbers.

The most common way to implement the generators is to enable initializing them *once* with an integer number, called the **seed** number. For a given seed number, the generator produces always the same sequence of numbers.

The seed number could be set from the system clock, or just selected manually. Doing the latter is actually almost always advisable, since this allows one to repeat the simulation identically, i.e. the simulation is **repeatable**.

- Science should per definition be repeatable!
- Repeatability is also useful from a practical point of view: can get more details if needed

The sequence in any pseudorandom generator will (has to) eventually repeat itself. The repeat interval (**period**) is an important measure of the quality of the generator.

- Period should be larger than amount of numbers needed!

The repeat interval in the most used decent generators is of the order of the number of integers that can be represented by a 4-byte integer, $2^{32} \approx 4 \times 10^9$. While this might seem large enough, it is not necessarily so!

- A simple calculation shows it is quite easy to exceed 4×10^9 random numbers nowadays.

Another important quantity to know for integer generators is the **maximum value** the routine can return. This is often (but not always) about the same as the maximum possible value of a (signed or unsigned) integer or long integer variable, i.e. e.g. $2^{31} - 1$ or $2^{32} - 1$.

To then get floating point random numbers divided evenly between 0.0 and 1.0 one should simply

convert the integer to a floating point number and divide it with the maximum m ,

$$r_{\text{float}} = \frac{r_{\text{integer}}}{m} \quad (1)$$

Random numbers divided evenly in an interval are called **uniform random numbers**.

Finally, it is often important to know exactly what the **limits** of the generator are. That is, for a floating point generator, can exactly 0 and 1 (within the numerical precision) be returned or not?

- Is the interval $[0, 1]$, $[0, 1[$, $]0, 1]$ or $]0, 1[$.

Purely mathematically this does not matter: the probability of hitting exactly the limit is exactly 0.

But on a computer it may **matter a lot!**. Especially if we work with 4-byte floating point values, which have a mantissa with only about 7 digit accuracy, the probability of hitting the limit is actually about 1 in 10 million. Even for 8-byte floating point variables, with a 15-digit mantissa, the probability of hitting the limit may not be negligible.

- It is easy to think of cases where this matters (imagine e.g. what happens if you take a logarithm of a random number which can be 0).
- Different applications need different limits, and many times it really does not matter after all.

4.4.1. The basic linear congruential algorithm

I will now attempt to take a pedagogical approach to generating random numbers, presenting one of the most common approaches step-by-step.

One of the simplest decent, and probably still most used method to generate random numbers is to use the following equation

$$I_{j+1} = aI_j + c \pmod{m} \quad (2)$$

Here “mod” is the modulus (remainder of division) operation.

This approach is called the **linear congruential algorithm** or if $c = 0$ the **multiplicative c. a.**

For the sake of example, let us take as the **seed** number $I_0 = 4$, and select the constants as $a = 7$, $c = 4$ and $m = 2^4 - 1 = 15$. Plugging in the numbers gives

$$aI_0 + c = 32$$

and after taking the modulus we get $I_1 = 2$. Continuing gives the following list of numbers:

Step i	I_i
0	4
1	2
2	3
3	10
4	14
5	12
6	13
7	5
8	9
9	7
10	8
11	0
12	4
13	2
14	3
15	10
16	14

So we see that $I_{12} = I_0$, i.e. the period is 12. This is pretty good, considering that obviously the period cannot exceed m .

Otherwise the sequence of numbers does indeed look fairly much random. But if we instead use e.g. $I_0 = 11$ the period becomes only 3:

Step i	I_i
0	11
1	6
2	1
3	11
4	6
5	1
6	11

This is of course *terrible!*

This example illustrates many important things:

1. It is not good to have a generator where the result depends on the seed number.

2. Hence care should be taking in selecting not only the random number generator but also the constants in it.

3. Do not fiddle around with the constants, thinking that a “random” change in one of the numbers would improve on the randomness of the results. Since the constants should be carefully selected, this is likely to lead to a much worse result

For the linear congruential generator there are actually quite simple rules which guarantee that the period reaches the maximum $m - 1$, i.e. we obtain all integers in the range $[0, m - 1]$ before repetition. These are:

- c and m should have no common prime factors
- $a - 1$ is divisible with all the prime factors of m
- $a - 1$ is divisible with 4 if m is divisible with 4.

Since the numbers chosen above, $a = 7$, $c = 4$ and $m = 15$, do not fulfill the second criterion, it is indeed to be expected that the full period is not achieved, and that the period length may depend on the seed. This is a terrible feature for a generator.

But if we had chosen $m = 17$, the full period will be achieved for any seed, except 5 (since $7 \times 5 + 4 \bmod 17 = 5$). In general, if a solution in the interval $[0, m - 1]$ exists for the equation $aI + c \bmod m = I$ there will always be one seed value I which fails in them.

But these criteria are **not enough to obtain an optimal generator**. Or in other words they are a necessary but not a sufficient set of rules. There may still be bad correlations in the numbers, and much more extensive testing should be carried out to obtain a truly good linear generator. Hence it is best not to fiddle around with the numbers on your own.

A note on coding generators in practice. There are two basic approaches one can use to code these generators. The first is to have two separate subroutines, one which sets the seed number, another which gives the random numbers. The first obviously takes the seed number as an argument, whereas the latter does not need any argument at all.

This is the approach taken e.g. in the C language. The system-provided random number generator is given a seed with the function

```
void srand(unsigned int seed)
```

and the actual random numbers are generated using the function

```
int rand(void)
```

This means that the subroutine `srand` has to pass the seed number to `rand()` somehow. But this can of course be easily implemented e.g. with an external variable, or a module in Fortran90. In C:

```
unsigned long seed=1;
```

```

int ansistandardrand(void) /* Not recommended for anything */
{
    long a=1103515245;
    long c=12345;
    long div=32768;

    seed = seed*a + c;
    return (unsigned int) (seed/65536) % div;
}

void ansistandardsrand(unsigned int seed_set)
{
    seed=seed_set;
}

```

Another approach is to require the seed number to always hang along in the function call, and be treated identically no matter what. In that case one simply sets seed once, then lets it change value without touching it outside the main routine. E.g.

```

seed=1;
for(i=0;i<=10000;i++) {

```

```

... (code not modifying seed) ...
randomnumber=rand(&seed)
... (code not modifying seed) ...
}

```

Which approach is better to use will of course depend on the other aspects of the code.

- The first one is slightly simpler in large codes, as one does not have to drag along the seed number over many places of the code.
- The latter one has the advantage that one could use different random numbers sequences at the same time easily, by using variables `seed1`, `seed2` and so on

But keeping the possible problems in mind still, the simple equation 2 is often quite good enough for many applications, especially with a careful choice of the parameters. A very widely used generator is the “Minimal standard” generator proposed by Park and Miller. It has

$$a = 7^5 = 16807 \quad c = 0(!) \quad m = 2^{31} - 1$$

This is by no means perfect, and indeed in many applications it is horribly bad.

When implementing it one meets a practical problem. Since the modulus factor m is almost 2^{31} , the values returned by the generator can also be close to 2^{31} . This means that on a computer one needs at least 31 bits to describe them. Then if we look at the product aI , one sees that the values can easily exceed 2^{32} .

Why is this a problem? Because many compilers to date can only handle integers with sizes up to 32 bits.

For instance on most 32-bit computers (including Pentiums) the sizes of the C integer variables are 16 bits (data type short) or 32 bits (data type int and long).

So doing the product above directly is not even possible on all machines.

Fortunately there exists a well-established solution by Schrage. It involves an approximate factorization of m , which we will not go into here. An implementation of this, from Numerical Recipes in C is:

```
#define IA 16807
#define IM 2147483647
```

```
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define MASK 123459876
```

```
float ran0(long *idum)
{
    long k;
    float ans;

    *idum ^= MASK;
    k=(*idum)/IQ;
    *idum=IA*(*idum-k*IQ)-IR*k;
    if (*idum < 0) *idum += IM;
    ans=AM*(*idum);
    *idum ^= MASK;
    return ans;
}
```

(if you want the actual integer seed number `idum` returned, just remove the multiplication with `AM`.)

This returns a random number in the interval $[0, 1[$, which is a common convention in the field.

The XOR function (\wedge) with MASK is just a trick to prevent problems if the routine is called with a seed of 0.

When using C, it is actually possible to avoid this implementation problem using a dirty trick.

- The C standard actually specifies that if you multiply two 32-bit integers and place the result into a 32-bit int register, the result returned is the low-order 32-bit of the real, up to 64-bit result.
- So this is exactly the operation of taking a modulus with 2^{32} for unsigned integers. Hence once can reduce the entire linear congruential generator into a single line:

```
unsigned long seed=1;  
...  
seed=1664525L*seed+1013904223L;  
...
```

The constants are by Knuth and Lewis [ref. in Numerical Recipes].

This is highly non-portable, since it requires that the length of “long” be 32 bits (not true on e.g.

Alphas). But it should not be much worse than the Park-Miller generator, and it does have the advantage of being extremely fast (only one multiplication and addition required), so its use might be justifiable in a temporary application which will never be transferred anywhere.

4.4.2. Problems with the linear congruential algorithm

The problems with the simple approach 2 can be divided into two categories: poor implementations, and fundamental problems.

4.4.2.1. Implementation problems

Although it might not seem necessary to list ways in which one can screw things up in programming, there is one important example which should be mentioned.

In the ANSI standard of the C language, the way to implement the system functions (in `stdlib.h`) `rand()` and `srand(seed)` is not specified. But they do specify an *example* of a generator, which is (here I have modified the code to add clarifying variable names) :

```
unsigned long seed=1;
int ansistandardrand(void) /* Not recommended for anything */
{
    long a=1103515245;
    long c=12345;
```

```

long div=32768;

seed = seed*a + c;
return (unsigned int) (seed/65536) % div;
}
void ansistandardsrand(unsigned int seed_set)
{
    seed=seed_set;
}

```

Note that the modulus m is not given explicitly, but really is 2^{32} . But then the returned value is divided by the very low number $\text{div}=32768$, meaning that only 32768 distinct values can be returned, and that the smallest non-zero value which can be returned is $1/32768$. Hence here the repeat interval is much longer than the number of returned values.

It is obvious that there are many many applications where values of less than $1/32768$ can be returned. So this example should essentially not be used for anything.

Unfortunately this generator, and even worse modifications of this, is in fact implemented on many compilers (both C and Fortran, don't know about Java). This leads us to the important rule of thumb: **never use the compiler standard random number generator!** And do not think that

even if you test on one system the compiler generator, that you can use it in many places. Since good codes should be portable to any system, this is not an acceptable line of thinking.

4.4.2.2. Fundamental problems

Some thought also reveals that there are quite basic fundamental problems associated with the linear congruential sequence generators.

One is that there is sequential correlation between successive calls. This can be a particular problem when generating manydimensional data

Related to this is this simple problem: if you consider a very small value of I in the Park-Miller generator

$$I_{j+1} = 16807I_j \quad (\text{mod } 2147483647)$$

say $I_j = 10$. Then $I_{j+1} = 168070$, i.e. still much less than the modulus 2147483647. So when I is divided by the modulus to give a floating-point value, we see that we first get 4.6566128×10^{-9} , then 4.6566128×10^{-8} . I.e. a very small value will *always* be followed by another small value, whereas for truly random numbers of course it could be followed by any number between 0 and 1 !

And yet another problem: the sequence can not return the same number twice after each other, even though in true random numbers even this should be possible.

4.4.3. Developments of the linear congruential algorithm

To overcome these problems, it seems like a pretty obvious idea to 'shuffle' the numbers somehow. This should at least solve the problem with successive small numbers and the 2D problem, and might help with the short-range correlation as well.

This can be simply achieved by having a table which holds a number of random numbers, and returns one of them in a random sequence. In practice, already a small table is enough to improve on the results significantly. Numerical Recipes presents a solution which has an array of 32 elements.

```
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
```

```

#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran1(long *idum)
{
    int j;
    long k;
    static long iy=0;
    static long iv[NTAB];
    float temp;

    if (*idum <= 0 || !iy) {
        if (-(*idum) < 1) *idum=1;
        else *idum = -(*idum);
        for (j=NTAB+7;j>=0;j--) {
            k=(*idum)/IQ;
            *idum=IA*(*idum-k*IQ)-IR*k;
            if (*idum < 0) *idum += IM;
            if (j < NTAB) iv[j] = *idum;
        }
    }
}

```



```

        iy=iv[0];
    }
    k=(*idum)/IQ;
    *idum=IA*(*idum-k*IQ)-IR*k;
    if (*idum < 0) *idum += IM;
    j=iy/NDIV;
    iy=iv[j];
    iv[j] = *idum;
    if ((temp=AM*iy) > RNMx) return RNMx;
    else return temp;
}

```

How does this work? The first long if clause initializes the sequence the first time the generator is called. It first generates 8 numbers which are thrown away, then fills in the array with 32 random number elements.

After this the ordinary Park-Miller generator follows, except that in the middle we have the operations

```

j=iy/NDIV;
iy=iv[j]
iv[j] = *idum

```

Here j first acquires a random value from the iy value, which is the random number generated in

the previous call. `NDIV` has a size such that `j` will be in the range 0-31, as it should. Then the returned random number is set from the stored number `iv[j]`, after which the random number generated in this call is written into the array in the position `j`.

This generator clearly solves many of the problems mentioned above. But even it can not return the same number twice after each other (in that case the period would be reduced to 32 or less).

4.4.4. Combined linear congruential generators

Proceeding in the development of the congruential generators, one can combine two single generators to form one with a very much longer period. This can be done by generating two sequences, then subtracting the result of one of them from the other (subtraction prevents an integer overflow). If the answer is negative, the number is wrapped to the positive side by adding the modulus m of one of the generators.

This forms a random-number sequence whose period can be the multiple of the period of the two generators. With generators similar to the Park-Miller generator with $m \sim 2^{31}$, one can thus reach a period of the order of $2^{62} \approx 4.6 \times 10^{18}$. Important in selecting the moduli m_1 and m_2 is that the periods they form do not share many common factors. In the following generator the periods are

$$m_1 - 1 = 2 \times 3 \times 7 \times 631 \times 81031 = 2147483562$$

and

$$m_2 - 1 = 2 \times 19 \times 31 \times 1019 \times 1789 = 2147483398$$

so they share only the factor of 2, and the period of the combined generator thus becomes $\approx 2.3 \times 10^{18}$. Thus at least period exhaustion is practically impossible on present-day computers, although not necessarily in 10 or 20 years.

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

float ran2(long *idum)
{
    int j;
```

```

long k;
static long idum2=123456789;
static long iy=0;
static long iv[NTAB];
float temp;

if (*idum <= 0) {
    if (-(*idum) < 1) *idum=1;
    else *idum = -(*idum);
    idum2=(*idum);
    for (j=NTAB+7;j>=0;j--) {
        k=(*idum)/IQ1;
        *idum=IA1*(*idum-k*IQ1)-k*IR1;
        if (*idum < 0) *idum += IM1;
        if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
}

k=(*idum)/IQ1;
*idum=IA1*(*idum-k*IQ1)-k*IR1;
if (*idum < 0) *idum += IM1;

```

```

! A
! A
! A

```

```

k=idum2/IQ2;                                ! B
idum2=IA2*(idum2-k*IQ2)-k*IR2;               ! B
if (idum2 < 0) idum2 += IM2;                  ! B
j=iy/NDIV;                                   ! S+G
iy=iv[j]-idum2;                              ! S+G
iv[j] = *idum;                               ! S+G
if (iy < 1) iy += IMM1;                       ! P
if ((temp=AM*iy) > RNMX) return RNMX;
else return temp;
}

```

What is going on here? The `if` clause is again initialization. Part A calculates the first random number using Schrage's method, part B likewise the second. Part S+G handles the shuffle and generates the combined random number, using the periodicity clause on line P.

Numerical Recipes places great trust in this algorithm, they even promise \$ 1000 to anyone who demonstrates that it fails in any known application. As far as I know, it has not been reported to fail so far, despite being subject to quite some testing [www.cs.adelaide.edu.au/users/paulc/papers/sccs-526/sccs-526.ps.gz]. (note that RAN2 in the first edition of numerical recipes is entirely different, and this has been reported to have problems [<http://www.lysator.liu.se/c/num-recipes-in-c.html>].)

4.4.5. Generalized feedback shift register algorithm

[Lewis, Payne, Journal of the ACM 20 (1973) 465]

Another, widely used and independent line of generators are the so called GFSR generators, which were originally, in 1973, developed to overcome some problems in the simplest linear congruential algorithms.

In this method, one starts with p random integer numbers $a_i, i = 0, 1, \dots, p - 1$ generated somehow in advance. Then the new elements k , with $k \geq p$, can be generated as

$$a_k = a_{k-p+q} \oplus a_{k-p}$$

where p and q are constants, $p > q$, and \oplus is the XOR logical operation.

(The XOR logical operation on two bits returns 1 if exactly one of the bits is 1, 0 otherwise.)

The difficult part is generating the initial numbers. In the original paper an older random number generator, so called Kendall sequence, was used to generate a first number of binary “1111100011011101010000100101100”, for $p = 5$. This number is the first number in a so called W sequence. The second number in the W is then the same but shifted forwards by 6 bits, i.e. 1011001111100011011101010000100’. Then the next number is again shifted forwards, and so on. This process gives the following numbers (the shift point is indicated by the horizontal line).

Index	Binary
0	1111100011011101010000100 101100
1	1011001111100011011101010 000100
2	0001001011001111100011011 101010
3	1010100001001011001111100 011011
4	0110111010100001001011001 111100

Note that one more shift would give back the original number, so no more shifts are useful.

Now the first five **columns** of this binary matrix is used as the originating sequence, i.e. we have the first 5 numbers as

Element	Binary
a_0	11010
a_1	10001
a_2	11011
a_3	11100
a_4	10011

This example is used for $p = 5$, $q = 2$. Let us look at in detail what happens after this. We have the numbers $k = 0, \dots, 4$ from above. We now want the number $k = 5$. Then from the definition of the algorithm

$$a_5 = a_{5-5+2} \oplus a_{5-5} = a_2 \oplus a_0 \quad (3)$$

The XOR operation can easily be done manually and gives

$$a_5 = 00001 \quad (4)$$

Then the next step would give

$$a_6 = a_3 \oplus a_1 = 01101 \quad (5)$$

and so on...

If we actually look at the original numbers in the first table we see these are just the columns of the number!

In algorithmic form the GFSR generator can be written as

- 1° $k=0$
- 2° Set a_0, \dots, a_{p-1} to some suitable initial values
- 3° Output a_k
- 4° $a_k = a_{k+q} \bmod p \oplus a_k$
- 5° $k = k + 1 \bmod p$
- 6° Goto step 3°

and as an actual C code (headers excluded)


```

main()
{
    unsigned int xn[5];
    unsigned int p,q;
    unsigned int k;

    p=5; q=2;

    /* GFSR generator of original example in Lewis/Payne paper */
    xn[0]=26;
    xn[1]=17;
    xn[2]=27;
    xn[3]=28;
    xn[4]=19;

    k=0;
    while(1) {
        printf("%d\n",xn[k]);
        xn[k]=xn[ ((k+q)%p) ] ^ xn[k];
        k=((k+1)%p);
    }
}

```

}

If we run this, we see that the period is 31. This actually follows from the full derivation and motivation: it can be shown that the period of the GFSR generator is

$$2^p - 1 \tag{6}$$

From this we also observe an obvious weakness of this original formulation: to get the initial values we actually already generated all the 31 independent numbers the method can produce.

- This is, however, not really a general problem: Lewis and Payne showed that more generally one needs to form a $p \times p$ matrix with linearly independent rows, and gave a recipe for how to do that.

With that approach, the method actually does have some clear advantages:

1. It is extremely fast after initialization: note above that a single loop step, without the print, only needs three memory references, one XOR and a couple of additions and moduli. All of these can easily be made in a single bit-level machine language operation (for instance a multiplication is on the bit-level waaay slower than an addition or logical operation). Hence it is well suited for situations where very fast generation of random numbers is needed with minimal demands on hardware, e.g. in small devices without too much computing power like mobile phones.

2. By increasing p one can rapidly increase the period.

The quality of the GFSR algorithms clearly depends on the choice of p and q . Like the parameters in the congruential algorithms, these have to be carefully chosen. The smallest ones are quite bad, but sequences where p is of the order of 1000 or more pass even quite advanced tests [Vattulainen, PRL 73 (1994) 2513].

The basic GFSR does have known weaknesses, the most obvious of which is that the schemes to initialize the array in a good way are complicated and quite slow. But there are many further developments of it, e.g. using several XOR operations, and non-linear versions. [<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/tgfsr3.pdf>]. We will discuss one of those later on.

4.4.5.1. Historical note

The original implementation of the XOR part (not the initialization of the method) is

```
FUNCTION RAND(M,P,Q,INTSIZ)
C
C M(P)=TABLE OF P PREVIOUS RANDCM NUMBERS.
C P,Q=POLYNOMIAL PARAMETERS:X**P+X**Q+1.
C .NOT. OPERATOR IMPLEMENTED IN ARITHMETIC.
C INTSIZ=INTEGER SIZE (BITS) OF HOST MACHINE: E.G.,
C IBM 360, 31; CDC 6000, 48; SRU 1100, 35; HP 2100, 15.
C
```

```

LOGICAL AA,BB,LCOMPJ,LCOMPK
INTEGER A,B,P,Q,INTSIZ,M(1)
EQUIVALENCE (AA,A),(BB,B),(MCOMPJ,LCOMPJ),(MCOMPK,LCOMPK)
DATA J/0/
N=(2** (INTSIZ-1)-1)*2+1
J=J+1
IF(J.GT.P) J=1
K=J+Q
IF(K.GT.P) K=K-P
MCOMPJ=N-M(J)
MCOMPK=N-M(K)
A=M(K)
B=M(J)
BB=LCOMPJ.AND.AA.OR.LCOMPK.AND.BB
M(J)=B
RAND=FLOAT(M(J))/FLOAT(N)
RETURN
END

```

which is pretty hard to decipher for anyone not grown up with 1960's Fortran... Note the extremely dirty trick of using EQUIVALENCE between logical and integer variables and the construction of an XOR operation by a combination of AND and OR. At that time Fortran did not have built-in bit-level logical operations, which lead to the above implementation. The authors simply *assumed* that a logical AND and OR also worked as bit-level operators on the whole word size. They then

tested it on 4 different computers of the day and concluded essentially “it seems to work fine on all of them” ...

4.4.6. Nonlinear generators

[<http://random.mat.sbg.ac.at/software/>; especially <ftp://random.mat.sbg.ac.at/pub/data/weingaThesis.ps>]

Many of the most modern and promising generators are based on the idea of using generators which have nonlinear properties.

4.4.6.1. Nonlinear congruential generators

The basic idea of the **inverse congruential generators** is to generate numbers using

$$y_{n+1} = a\bar{y}_n + b \pmod{M}$$

This at first looks exactly like the equation for the ordinary linear congruential generators. But the difference is given by the bar on y . This signifies the solution for the equation

$$c\bar{c} = 1 \pmod{M}$$

where c is given and \bar{c} is the unknown. So instead of directly calculating the new number from y_n , one first calculates the congruence \bar{y}_n .

Calculating \bar{c} is actually not quite easy, and won't be discussed here. The method to do can be found among the software given on the link above (look for the `prng-3.0.tar` library and subroutines `prng_inverse_*` there).

This basic generator is already clearly better than the ordinary linear generators when the constants and modulus are comparable. It has been further developed by replacing the \bar{y} operation with other operators.

Another development is simply to add a quadratic term to the linear generator to produce the **quadratic congruential generator**,

$$y_{n+1} = ay_n^2 + by_n + c \pmod{M}$$

Several other developments of generators along this line exist.

4.4.6.2. Nonlinear GFSR generators

Matsumoto and Kurita have recently (1990's) developed the basic GFSR generator further. The basic change is called a “twist”, i.e. instead of using the simple GFSR operation of

$$a_k = a_{k-p+q} \oplus a_{k-p} \tag{7}$$

one uses

$$a_k = a_{k-p+q} \oplus a_{k-p}A \quad (8)$$

where A is a binary matrix of dimensions $w \times w$ where w is the word size (e.g. 32 for an unsigned 4-byte integer). a_{k-p} must here be considered a row vector for the vector \times matrix operation to make sense.

This is a “twisted GFSR” or TGFSR generator. [Matsumoto and Kurita, 1992 ACM transactions on modelling and Computer Simulations 2 (179-194); <http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/tgfsr3.pdf>]. With suitable choices of p , q and A this generator can achieve the maximal period of

$$2^{pw} - 1 \quad (9)$$

which is about a factor of 2^w more than the period of the basic GFSR. This is a huge improvement, of course; even for 4-byte integers it is 2^{32} more.

An additional major advantage of the TGFSR is that A can be chosen to make the algorithmic implementation almost as simple as that of the basic GFSR. That is, the algorithm can be identical to the GFSR one given above except that step 4 changes. The TGFSR algorithm is

- 1° $k=0$
- 2° Set a_0, \dots, a_{p-1} to some suitable initial values
- 3° Output a_k
- 4° $a_k = a_{k+q \bmod p} \oplus \text{shiftright} a_k \oplus \begin{cases} 0 & \text{if least significant bit of } a_k = 0 \\ \mathbf{a} & \text{if least significant bit of } a_k = 1 \end{cases}$
- 5° $k = k + 1 \bmod p$
- 6° Goto step 3°

Here \mathbf{a} is a fixed constant binary vector of size w and the shift means a one-bit shift operation.

The TGFSR generator is proven to have several major advantages over the GFSR and pass quite advanced tests. One obvious practical advantage is that the number of initialization vectors p does not need to be very large to achieve a long period. E.g. already $w = 16, p = 25, q = 11, \mathbf{a} = A875$ (last one is hexadecimal) gives a period of $2^{400} - 1$.

- Moreover, also the initialization vector is much less sensitive than in the GFSR, any initialization vector (except all zero) will in fact do since the algorithm in any case goes through the full possible set of ways to obtains sequences of p integers from the set of 2^w possible integers.

A further development of the TGFSR is the **Mersenne Twister** from 1996-1997. It is based on the same idea as the TGFSR, the difference is that now the crucial step is

$$a_k = a_{k-p+q} \oplus (a_{k-p}^u | a_{k-p+1}^t) A \quad (10)$$

where a_{k-p}^u denotes the upper $w - r$ bits of the vector a_{k-p} , and a_{k-p+1}^t the lower r bits of the vector a_{k-p+1} . That is r is now an extra constant between 0 and w . The alert reader will notice that if $r = 0$ this reduces back to the TGFSR.

- The A can still be handled in the same manner as in the TGFSR, and initialization is equally easy.

With suitable good choices of the parameters this generator can achieve a period of

$$2^{pw-r} - 1 \quad (11)$$

The values recommended by the authors are $(w, p, q, r) = (32, 624, 397, 31)$ and $\mathbf{a} = 9908B0DF$, which gives the period

$$2^{19937} - 1$$

which should be enough for a few years to come... The name “Mersenne” comes from the choice of w, p and r to give a period which is a so called Mersenne prime number.

In addition to being proven to have many good features, this generator also has the advantage of being readily available in source code from the web, see <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

The Mersenne twister has passed a wide range of tests on random numbers, and is despite its fairly complex nature very efficient computationally; sometimes even faster than the standard C language generator.

4.4.7. Combined generators

It is of course also possible to combine generators of different type, which when carried out well should minimize the chances of the algorithm of one generator messing things up. One famous generator which seems to be generally held in good respect is the RANMAR generator by G. Marsaglia (source code with built-in documentation `ranmar.f` available on the course home page).

It has a period of $\approx 2^{144}$ which also should be good enough for some time to come...

4.5. Tests of generators

4.5.1. Theoretical tests

[Mainly from G+T 12.19, also parts from Knuth]

We have already seen several very basic tests of algorithms, but we will reiterate what they are.

0. The very first thing to consider is that the period of the generator is larger than the number of random numbers needed in your simulations. Testing for this simply amounts to finding or knowing the period.

1. Another very basic test for a generator is of course to check that it produces a uniform distribution between 0 and 1 in 1D. But failing this test is so basic that no sensible bug-free generator does it.

2. Many-dimensional smoothness tests. As an extension of the above, an important test is that the random-number distribution is flat also in many dimensions. Here actually many generators already start to have problems

Testing for problems 1 and 2 is simple, just doing the plots is the basic tests in 1, 2 and 3D. Zooming in on parts of the test region may be useful in case the overall region looks OK.

3. χ^2 test

One basic test for generators which have passed test 1, is to repeat the test but look at the fluctuations from the average value, when one generates some finite number of N random numbers.

- I.e. if we say collect generate 100 random numbers between 0 and 1 and make statistics of them in bins of width 0.1, one should not normally get exactly 10 in every bin, but say 8 in one, 13 in the next and so on.
 - One can then look at the fluctuations, and find whether they are what is expected from probability theory.

If we consider generating N random number and placing them in M bins, then it is clear that the expected value E_i for bin i is $E_i = N/M$. If the observed value in each bin is y_i , then we can calculate the χ -square statistic of this test with

$$\chi^2 = \sum_{i=1}^M \frac{(y_i - E_i)^2}{E_i}$$

For large values of M ($M > 30$) there should be a 50 % probability that $\chi^2 > M - 2/3$, and a 50 % probability that $\chi^2 < M - 2/3$. Now we can test the generator by calculating χ^2 numerous

times (for different random number sequences of course), and seeing whether it has the average $M - 2/3$.

It is also possible to predict the probabilities for different percentage points, and then calculate how often χ^2 exceeds this point. For instance, for $M = 50$ χ^2 should exceed $1.52M$ at most 1 % of the time. This may actually be a much stronger test of whether the distribution is what it should be.

(*n.b.* Gould-Tobochnik has an error in their equation 12.19.c : they say $\chi^2 \leq M$, when it should be $\chi^2 \approx M$.)

Finding **correlation problems** may be difficult. We described the “two small numbers in sequence” problem above, and mentioned that other exist. In fact, the most complex correlation tests are the empirical tests mentioned in the next section.

4. Autocorrelation tests

One way to look for short-range correlations in a sequence of random numbers x_i is to use the autocorrelation function

$$C(k) = \frac{\langle x_{i+k}x_i \rangle - \langle x_i \rangle^2}{\langle x_i x_i \rangle - \langle x_i \rangle^2}$$

where $\langle x_{i+k}x_i \rangle$ is found by forming all possible products $x_{i+k}x_i$ for a given k and dividing by the number of terms in the product. $C(k)$ should become zero when $k \rightarrow \infty$.

4.5.2. Empirical tests

The tests described above were of a rather general nature, and a good generator should pass them all. However, even these tests do almost certainly not prove that a generator is good enough.

A logical continuation of testing is to use tests which are close to the system studied. For instance, if one simulates the Ising model, a good way to test the generator is to simulate the Ising model in a case where the accurate answer is known. These tests are called **empirical, physical** or **application-specific**.

Here is a quick overview of some of them;

1. S_N test. The test uses random walkers on a line and calculates the total number of sites visited by the random walkers (versus the number of jumps made). In the test, N random walkers move simultaneously without any interaction such that, at any jump attempt, they can make a jump to the left or to the right with equal probability. After $n \gg 1$ jumps by all random walkers, the

mean number of sites visited $S_{N,n}$ has an asymptotic form

$$S_{N,n} \sim \log N^{1/2} n^x$$

where the exponent x should be $1/2$ according to theory. The value of the exponent x observed from simulations serves as a measure of correlations.

2. Interface roughening in 1+1 dimensions. In this case the roughening of a 1-dimensional interface is followed with time. Consider two (independent) 1D random walks, which determine the heights $h_1(n)$ and $h_2(n)$ of two interfaces versus the number of jumps made n . The height of the interface between the two random walkers is then

$$h_1(n) - h_2(n),$$

whose height-height correlation function follows a power law in distance n where the roughening exponent should be $1/2$.

3. Ising model autocorrelation test. In this test, averages of some physical quantities such as the energy, the susceptibility, and the updated cluster size in the 2D Ising model are calculated. Additionally, their autocorrelation functions and corresponding integrated autocorrelation values are

determined. The exact value is known only for the energy; for the other quantities, the test works by comparing results of different generators.

This is a fairly stringent test; e.g. RAN3 from Numerical Recipes first edition fails this test, as do several of the GFSR generators.

4.5.3. So, what generator should I use?

To summarize all of this discussion, I give my personal view of what generator to use when.

I can see almost no reason to ever use anything less than the Park-Miller minimal generator. And since even this has many known problems, it should be used only in cases where the random numbers are of secondary importance. This can be for instance when only a few hundred random numbers are needed for e.g. selecting impact points on a 2D surface, or when initializing velocities of atoms in an MD simulation.

In cases where random numbers are used all through a large simulation run in crucial parts of the code, I would recommend using something better than the Park-Miller generator.

Since there are no guarantees any generator is good enough for a problem which has not been studied before, a good strategy would be to choose a few completely different generators and repeat some of the central simulations with all of these. If no dependence on the choice of generator is

found, there *probably* is no problem. The generators chosen could be e.g. the RAN2 from numerical Recipes second edition, the Mersenne twister and RANMAR.

(And remember that if you find problems with RAN2, you can claim the \$1000 reward!)