# 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. The first concept can simplistically be defined to mean that there is no way to predict what the next random number is (short of being God or having invented a time machine). The second means 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), but which still appear random for all practical purposes. The third means numbers which act as random numbers in some sort of simulations, but are well-ordered in some other types. More precise explanations for the latter 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. Because of the fallibility and slowness of human beings, this is a truly non-recommendable way for generating larger sets of random numbers. But it is widely used for selecting the seed number (see below) for electronic random number generators. Since for a good generator all seeds are equal, this is acceptable as long as you do not write very many seeds for a single problem (in which case human correlations could become a problem).

---

Another one, which was historically used to some extent, and perhaps still is, is to select numbers from some number sequence, e.g. the phone book or the decimals of $\pi$. 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 $\pi$ are not supposed to have correlations (although I dot know whether this is really true for the most stringent tests).

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

3.14159265358979323846264338327950288419716939937510 5

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 simplest are of course those akin to the ones used in lottery or roulette, but 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. The only processor I know myself was the music processor of the good old Commodore 64, which had one byte which gave random bits (although to be honest I am not sure exactly how that worked).

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. This actually did happen on the Commodore 64 (EXAMPLE TOLD DURING LECTURE).

---

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 essentially 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

### 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 it would use a pseudorandom number generator where the seed could be somehow figured out, 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 (sort of an uncertainty principle in processors: the measurement itself changes the result), 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 (whatever that is).

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

---

The downside here is that the implementation can not possibly be fully portable. As of the writing of this (Jan 2004), the routine works for Sun UltraSparc's, Pentium II, III, 4, Celeron, Athlon and Duron, Itanium, and PowerPC G4. A nice list, but by no means complete.

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

[http://www.physics.helsinki.fi/∼vattulai/rngs.html. Numerical Recipes. Knuth. Karimäkis notes]

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.** This is very important both from a basic science philosophy point of view - any real science needs to be reproducible. And also from a practical point of view; often one wants to repeat a simulation where something interesting happened e.g. to enable printing more detailed output from the interesting section.

---

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. Obviously, the period of the generator should be larger than the number of random numbers used in one simulation, to avoid the possibility that the random numbers cause distortions to the answer.

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. Remember that present day computers can handle of the order of $10^9$ operations per second. If the innermost loop of a simulation algorithm uses one random number and say 100 operations per loop step, one will have used up the $4 \times 10^9$ independent random numbers in 400 seconds. This is nothing in computational physics or chemistry, where a single run can easily go on for days or even weeks.

---

The final 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 $m$,

$$r_{\mathrm{float}} = \frac{r_{\mathrm{integer}}}{m} \tag{1}$$

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

### 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

$$I_{j+1} = aI_j + c \qquad (\mathrm{mod}\ m) \tag{2}$$

Here "mod" us 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 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 (this will be illustrated in the exercises)

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 $7 \times 5 + 4 \mathrm{mod}\ 17 = 5$). In general, if a solution in the interval $[0, m-1]$ exists for the equation $aI + c \ (\mathrm{mod}\ m) = I$ 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 let's 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 (this might be needed e.g. if one wants to have one repeatable sequence of random numbers initialized by hand, and one non-repeatable sequence initialized by the system clock every time).

---

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

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

This is by no means perfect, and indeed in many applications it is horribly bad. But in most cases it is good enough.

---

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 most 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;
}
```

This actually returns a random number in the interval $[0.0, 1.0[$, which is a common convention in the field (if you want the actual integer seed number `idum` returned, just remove the multiplication with `AM`.

The XOR function (^ ) 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 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 an implementation, 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 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 may be 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 any good code should be portable to any system, this is not an acceptable line of thinking.

(Just out of curiosity, I looked through what the `glibc` C libraries (used in Linux) actually contain today. The version was 2.2.4. The routine `rand()` seems to be based on the Schrage routine, but uses some sort of complex state mixing scheme to improve on the period. The period is $2.88 \times 10^9$, i.e. quite acceptable. But calling the routine is very slow, which is not good either.

### 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 (an example is given in the exercises); the data may appear as planes in the manydimensional space.

---

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

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

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 $4.6566128 \times 10^{-9}$, then $4.6566128 \times 10^{-8}$. 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.

---

There are also more subtle problems, such as so called short-range serial correlations. And they also often can have really terrible problems in a 2D plane; this is illustrated in an exercise.

### 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 (a periodic boundary condition in 1D)

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

$\approx 2.3\times10^{18}$. 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;                                  ! A
```

```
        *idum=IA1*(*idum-k*IQ1)-k*IR1;                  ! A
        if (*idum < 0) *idum += IM1;                    ! 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 $a_i, i = 0, 1, \ldots, p-1$ 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 is 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, so called Kendall sequence, was used to generate a first number of binary "11111000110111010100001001011 00", 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. 10110011111000110111010100001 00'. 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 | 111110001101110101000010\|0101100 |
| 1 | 101100111110001101110101\|0000100 |
| 2 | 000100101100111110001101\|1101010 |
| 3 | 101010000100101100111110\|0011011 |
| 4 | 011011101010000100101100\|1111100 |

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 $k = 0, \ldots, 4$ from above. We now want the number $k = 5$. Then from the definition of the algorithm

$$a_5 = a_{5-5+2} \oplus a_{5-5} = a_2 \oplus a_0 \tag{3}$$

The XOR operation can easily be done manually and gives

$$a_5 = 00001 \tag{4}$$

Then the next step would give

$$a_6 = a_3 \oplus a_1 = 01101 \tag{5}$$

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 $a_0, \ldots, a_{p-1}$ 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 $3^\circ$

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 multiply 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,O=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

$$y_{n+1} = a\bar{y}_n + b \qquad (\text{mod } M)$$

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

$$c\bar{c} = 1 \qquad (\text{mod } M)$$

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_`$\star$ there).

This basic generator is already clearly better than the ordinary linear generators when the constants 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**,

$$y_{n+1} = ay_n^2 + by_n + c \qquad (\text{mod } M)$$

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

$$a_k = a_{k-p+q} \oplus a_{k-p}A \tag{8}$$

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 Similations 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

$$2^{pw} - 1 \tag{9}$$

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 $a_0, \ldots, a_{p-1}$ to some suitable initial values

**3°** Output $a_k$

**4°** $a_k = a_{k+q \bmod p} \oplus \mathrm{shiftright} a_k \oplus \begin{cases} 0 & \text{if least significant bit of } a_k = 0 \\ \mathbf{a} & \text{if least significant bit of} a_k = 1 \end{cases}$

**5°** $k = k + 1 \bmod p$

**6°** Goto step $3^\circ$

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 $w = 16, p = 25, q = 11, \mathbf{a} = A875$ 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 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

$$a_k = a_{k-p+q} \oplus (a^u_{k-p} | a^t_{k-p+1}) A \tag{10}$$

where $a^u_{k-p}$ denotes the upper $w - r$ bits of the vector $a_{k-p}$, and $a^t_{k-p+1}$ 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

$$2^{pw-r} - 1 \tag{11}$$

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/eindex.html.

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

### 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, as we shall see in the exercises.

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.** $\chi^2$ 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. (FIGURE DRAWN DURING LECTURE) 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 $\chi$-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 $\chi^2$ 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 $\chi^2$ exceeds this point. For instance, for $M = 50$ $\chi^2$ 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 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 $k \longrightarrow \infty$.

### 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.**

---

The research group of Vattulainen, Ala-Nissilä and Kankaala (formerly at HIP, now at HUT) have developed several such tests. Here is a quick overview of some of them; codes to test them can be found in http://www.physics.helsinki.fi/∼vattulai/rngs.html.

---

**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 $n >> 1$ 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.f. 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!)

# 4.6. Generating non-uniform random numbers

[Numerical Recipes, Karimäki lecture notes]

So far we have only discussed generating random numbers in a uniform distribution, but at quite some length. There is a good reason to this - random numbers in any other distribution are almost always generated starting from random numbers distributed uniformly between 0 and 1.

But in the natural sciences it is clear that data often comes in many other forms. E.g. the peaks in $\gamma$ spectra have a Gaussian or Lorentzian shape, the decay activity of a radioactive sample follows an exponential function, and so on.

There are two basic approaches to generate other distributions than the uniform. The first is the analytical one, the other the numerical von Neumann rejection method.

## 4.6.1. Analytical approach (inversion method)

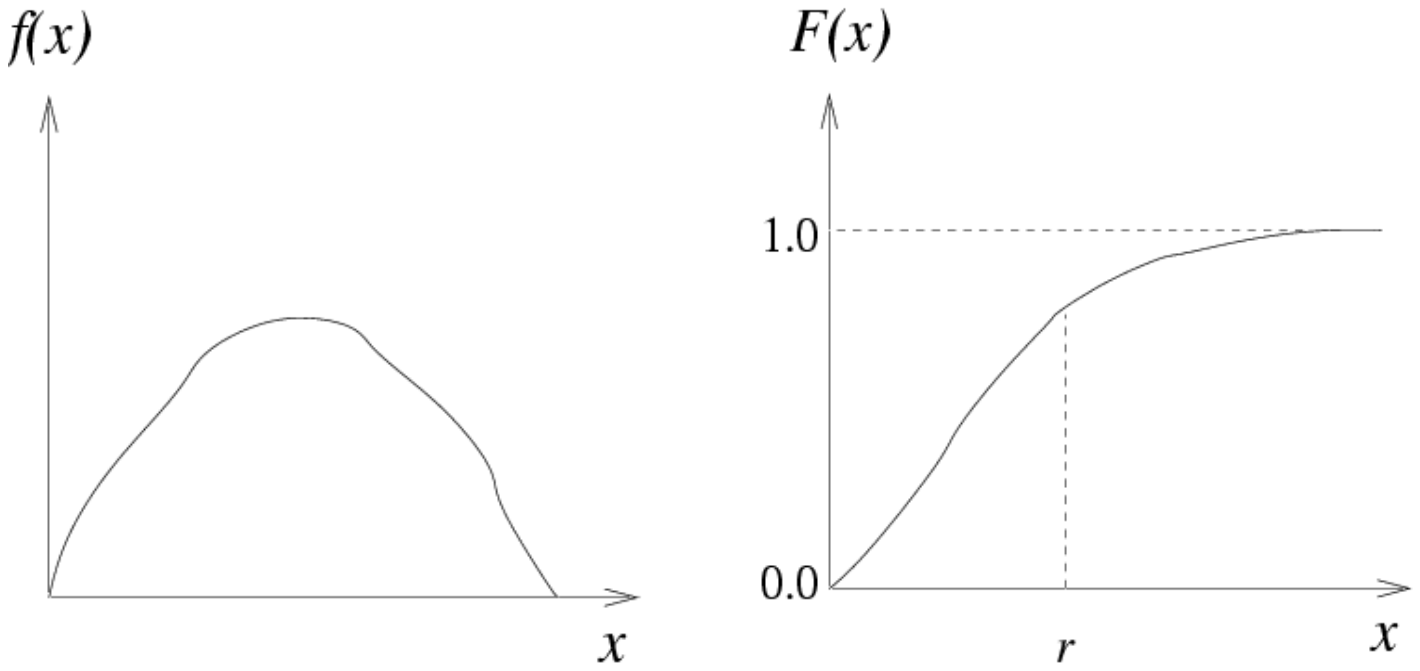
We want to calculate random numbers which are distributed as some arbitrary function $f(x)$. To be a reasonable probability distribution, the function must have the properties

$$f(x) > 0 \text{ for all } x$$

and

$$\int_{-\infty}^{\infty} f(x)dx = 1$$

Otherwise there are no limits on what $f$ can be.



In the derivation, we will also need the cumulative distribution function ("kertymäfunktio")

$$F(x) = \int_{-\infty}^{x} f(t)dt$$

and its inverse function $F^{-1}(s)$,

$$s = F(x) \Longleftrightarrow x = F^{-1}(s)$$

Let us denote our generator for uniform random numbers $P_u(0,1)$. We now postulate that to generate random numbers $r$ distributed as $f(x)$, we should perform the following steps:

**1°** Generate a uniformly distributed number $u = P_u(0,1)$

**2°** Calculate $x = F^{-1}(u)$

---

To prove this, we will show that the cumulative distribution function $F'(x)$ of the numbers $x$ is the function $F(x)$. Since each function $f$ has a unique cumulative function, this is enough as a proof.

Consider the probability that a point $x$ is below the given point $r$,

$$F'(r) = P(x \leq r)$$

This is the cumulative distribution function of the $x$, but we do not yet know what the function is. But now, using (2° ),

$$F'(r) = P(x \leq r) = P(F^{-1}(u) \leq r)$$

Now we can apply the function F on both sides of the inequality in the parentheses, and get

$$F'(r) = P(F(F^{-1}(u)) \leq F(r)) = P(u \leq F(r))$$

But because $u$ is just a uniformly distributed number, we have simply $P(a \leq b) = b$ and hence

$$F'(r) = P(u \leq F(r)) = F(r) \qquad \text{q.e.d.}$$

---

So the algorithm is very simple, but requires that it is possible to calculate the inverse function of the integral of the function we want. Since all functions are not integrable, using this analytical approach is not always possible.

Let us illustrate how this works in practice with an example. Say we want to have random numbers with an exponential decay above 0, i.e.

$$f(x) = \begin{cases} e^{-x} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Now we first calculate

$$F(x) = \int_0^x f(t)dt = \int_0^x e^{-x} = 1 - e^{-x}$$

and then solve

$$s = F(x) = 1 - e^{-x} \Longrightarrow x = -\log(1-s) \; i.e. \; F^{-1}(u) = -\log(1-u)$$

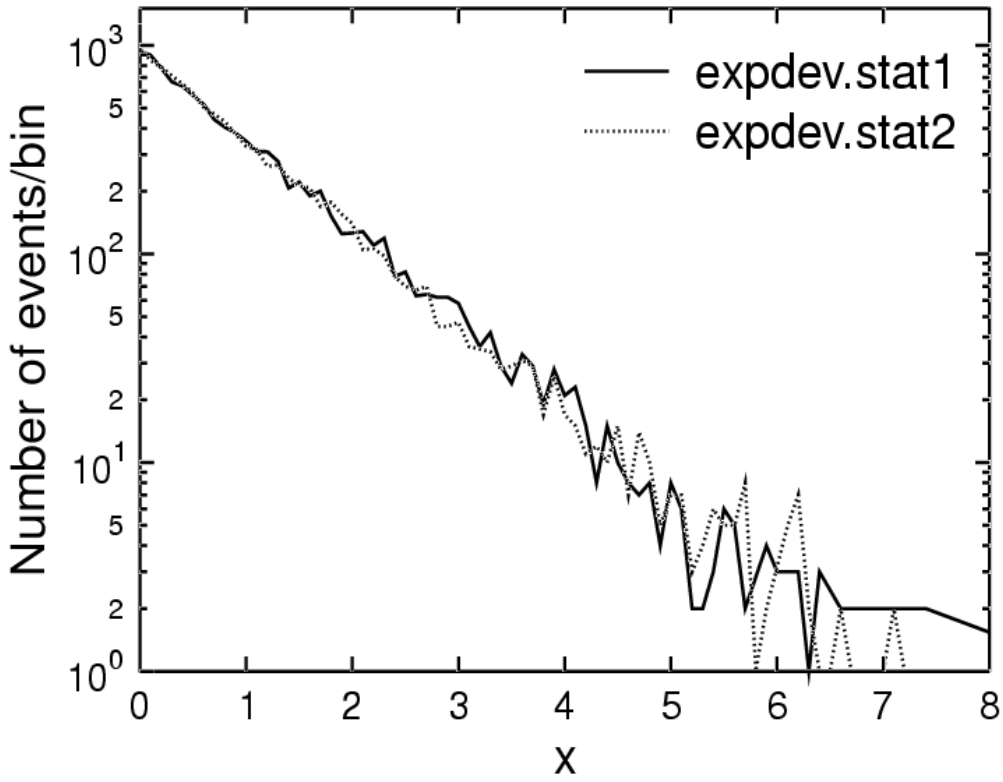
But because $u$ is a random number between 0 and 1, so is $1 - u$, and we can reduce this to

$$F^{-1}(u) = -\log(u)$$

To test whether this really works, I wrote the following small gawk script (since this is for demo only, it is excusable to use the system random number generator):

```
gawk 'BEGIN {
        # Initialize random number generator from system clock
        srand();
        # Generate 10000 exponential deviates:
        for(i=0;i<10000;i++) {
                print -log(rand());
        }
        exit;
}'
```

The statistics of this is, from two runs:




((Hint for those of you using Unix: a single command line is enough to generate simple statistics like this:

```
expdev | awk '{ printf "%.1f\n",int($1*10)*0.1; }' | sort -n | uniq -c |
awk '{ print $2,$1 }' > expdev.stat1
```

))

So it really is a nice exponential dependence (since it looks linear on a log scale).

#### 4.6.1.1. Discrete distribution

For a discrete distribution, we can also use the inversion method. Let us say we have points $p_i, i = 1, \ldots, N$ which define the probability distribution function for evenly some set of points $x_i$. We then have to generate the cumulative distribution function as a discrete function $F_j$, which can be achieved by summation:

$$F_j = \sum_{i=1}^{j} p_i$$

for all $j = 1, \ldots, N$. We further have to set $F_0 = 0$ and ensure that $F_N = 1$ to handle the ends correctly. Then the generation algorithm becomes

**1°** Generate a uniformly distributed number $u = P_u(0, 1)$

**2°** Find $k$ such that $F_{k-1} < u \leq F_k$

which gives integers $k$ whose probability of occurring is proportional to $p_k$.

Step 2° is very easy to do e.g. using a binary search.

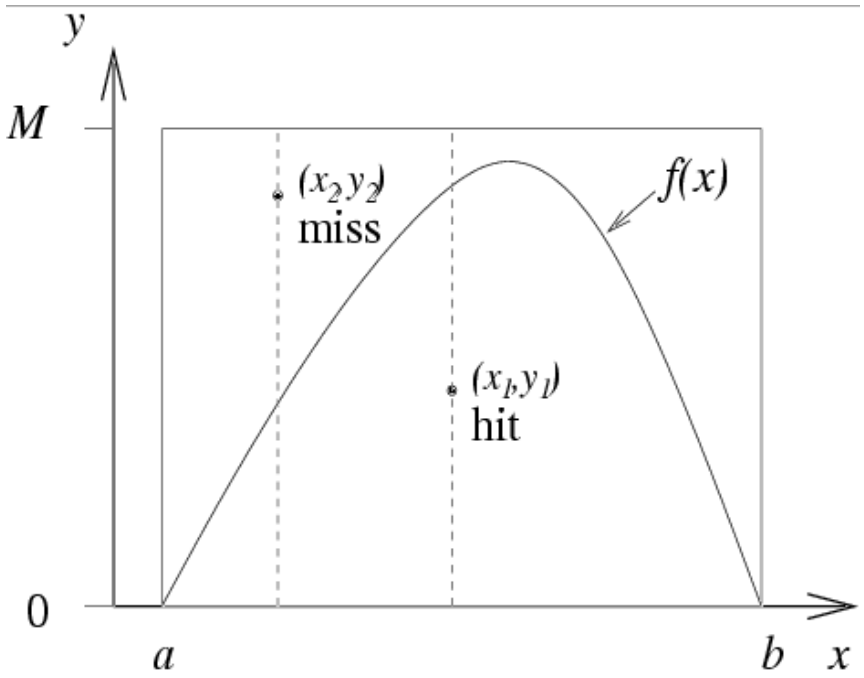
Note that in case it is not possible to find an inverse $F^{-1}(x)$ for an analytical function, one possibility is to tabulate it as a discrete distribution, then use the method described here, and then interpolate to get a somewhat more accurate estimate than given by the discrete distribution (which produces a stepwise function). The interpolation could be linear or even using splines if the best possible accuracy is desired.

Also note that if the set of points $p_i$ has a region where it is zero in the middle of the distribution, $F_j$ will be flat in this region. If you use interpolation schemes when doing step $2^\circ$, you may still create a point in the forbidden region, which may well lead to either your program crashing, or (even worse!) completely unphysical results. So be careful. [K. Arstila, private communication].

### 4.6.2. von Neumann rejection method

There is another way to generate random numbers, which is purely numerical and works for any finite-valued function in a finite interval, regardless of whether it can be integrated or inverted. It is called the (von Neumann) **rejection** method or **hit-and-miss** method

The idea is straightforward. Consider a function $f(x)$ defined in some finite interval $x \in [a, b]$. It has to be normalized to give probabilities. Let $M$ be an number which is $\geq f(x)$ for any $x$ in the interval:




Now a rather obvious algorithm to generate a random number in this distribution is

**1**° Generate a uniformly distributed number $x = P_u(a, b)$

**2**° Generate a uniformly distributed number $y = P_u(0, M)$

**3**° If $y > f(x)$ this is a **miss:** return to 1°

**4**° Otherwise this is a **hit:** return $x$

This way we obtain random numbers $x$ which are distributed according to the given distribution $f(x)$. Note that $y$ only carries the role of a checking variable and is not returned.

---

This seems nice and easy. The downside is of course that we do some redundant work: all the "miss" numbers were generated in vain. The probability to get a hit is

$$P(\text{hit}) = \frac{\int_a^b f(x)dx}{M(b-a)} = \frac{1}{M(b-a)}$$

For a function like that plotted in the figure above, this is not too bad. But if the function is highly peaked, or has a long weak tail, the number of misses will be enormous.

Consider for instance an exponential function $e^{-x}, x > 0$. If the problem is such that one can neglect very small values, one can use some cutoff value $b >> 1$ in the generation of random numbers. One could then use the hit-and-miss algorithm in the interval $[0, b]$ to generate random numbers for the exponential function. The normalized function would be

$$f(x) = \frac{e^{-x}}{\int_0^b e^{-x}dx} = \frac{e^{-x}}{1-e^{-b}}, \qquad x \in [0, b]$$

and the probability of misses

$$P(\text{miss}) = 1 - \frac{1}{M(b-0)} = 1 - \frac{1}{Mb}$$

and since the maximum of $f(x)$ is at 0, we can use

$$M = \frac{1}{1 - e^{-b}}$$

and get

$$P(\text{miss}) = 1 - \frac{1 - e^{-b}}{b}$$

If for instance $b = 100$, we have $\approx 99$ % misses, i.e. terrible efficiency. Obviously it would be much better in this case to use the analytical approach.

### 4.6.3. Combined analytical-rejection method

Unfortunately for many functions it will not be possible to do the analytical approach. But there may still be a way to do better than the basic hit-and-miss algorithm.

In case the shape of the function is known (which certainly almost always is the case in 1D), then

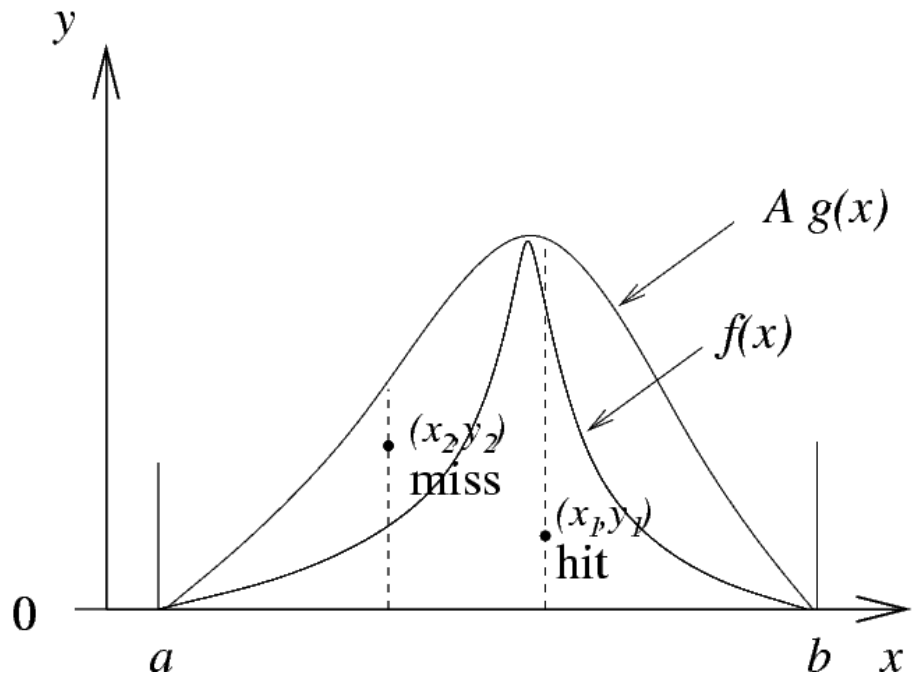
maybe we can find a function which is always larger than the one to be generated, but only slightly so. Then if we can generate analytically random numbers for the larger function, we can again use the hit-and-miss method, but with much less misses.

---

To put this more precisely, say we can find a function $g(x)$ for which a constant $A$ exists such that

$$Ag(x) \geq f(x) \text{ for all } x \in [a, b].$$

It is important to include the constant $a$ here because both $g(x)$ and $f(x)$ are probabilities normalized to one. For this to be useful, we further have to demand that it is possible to form the inverse of the cumulative function $G^{-1}(x)$ of $g(x)$.




Then the algorithm becomes:

**1°** Generate a uniformly distributed number $u = P_u(0, 1)$

**2°** Generate a number distributed as $g(x)$: $x = G^{-1}(u)$

**3°** Generate a uniformly distributed number $y = P_u(0, Ag(x))$

**4°** If $y > f(x)$ this is a **miss:** return to $1^\circ$

**5°** Otherwise this is a **hit:** return $x$

Note that we do not ever have to generate numbers in the interval $[a, b]$ in this case, so the limits can be at infinity as well. This can be a major advantage over the pure hit-and-miss method, which requires a finite cutoff to be used for functions extending to infinity.

### 4.6.4. Generating Gaussian random numbers

The Gaussian function

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

is of course one of the most common functions in science, and there are numerous applications where one wants to generate random numbers distributed as $f(x)$. Unfortunately it is not integrable, as we all know, so using the inversion method directly is not possible.

---

But as you probably all also remember, the definite integral from $-\infty$ to $\infty$ of $f(x)$ can be evaluated by a trick using two dimensions. For simplicity, let's work with the the Gaussian distribution in the form centered at $0$ with $\sigma = 1$,

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

The integral can be calculated by taking the square of the integral

$$\left[\int_{-\infty}^{\infty} e^{-\frac{1}{2}x^2} dx\right]^2 = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-\frac{1}{2}x^2} e^{-\frac{1}{2}y^2} dxdy = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-\frac{1}{2}(x^2+y^2)} dxdy$$

and switching to polar coordinates $r^2 = x^2 + y^2$, $dxdy = rdrd\phi$:

$$= \int_0^{\infty}\int_0^{2\pi} e^{-\frac{1}{2}r^2} rdrd\phi = 2\pi \int_0^{\infty} e^{-\frac{1}{2}r^2} d\left(\frac{1}{2}r^2\right) = 2\pi$$

---

The **Box-Muller method** to generate random numbers with a Gaussian distribution relies on a similar trick.

In this method, we also consider two Gaussian distributions in 2D. Their joint distribution is

$$f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2} = \frac{1}{2\pi} e^{-\frac{1}{2}(x^2+y^2)}$$

Switching again to polar coordinates, and remembering that the surface element transforms as

$dxdy = rdrd\phi$ we get the polar density distribution function

$$g(r,\phi) = f(x,y)r = \frac{1}{2\pi} r e^{-\frac{1}{2}r^2}$$

We can now separate the $r$ and $\phi$ contributions:

$$g(r,\phi) = f_\phi(\phi) f_r(r)$$

where

$$f_\phi(\phi) = \frac{1}{2\pi}$$
$$f_r(r) = r e^{-\frac{1}{2}r^2}$$

So if we can generate $f_\phi$ and $f_r$ separately, we will also be able to generate the joint 2D distribution and hence two Gaussian numbers at a time.

Generating $f_\phi$ is trivial, we just need to form a uniform number and multiply by $2\pi$ to get an even distribution in the range $[0, 2\pi]$, which has the value $\frac{1}{2\pi}$ everywhere. $f_r$ can also be handled since

$$F_r(r) = \int_0^r r e^{-\frac{1}{2}r^2} = 1 - e^{-\frac{1}{2}r^2}$$

which can be inverted to give

$$F_r^{-1}(u) = \sqrt{-2\log(1-u)}$$

So we can obtain both $r$ and $\phi$. After this the $x$ and $y$ can be obtained easily using

$$\begin{cases} x = r\cos\phi \\ y = r\sin\phi \end{cases}$$

So the **polar** or **Box-Muller** algorithm becomes

**1°** Generate a uniformly distributed number $u_1 = P_u(0,1)$

**2°** Calculate $\phi = 2\pi u_1$

**3°** Generate a uniformly distributed number $u_2 = P_u(0,1)$

**4°** Calculate $r = \sqrt{-2\log u_2}$

**5°** Obtain $x$ and $y$ using

$$\begin{cases} x = r\cos\phi \\ y = r\sin\phi \end{cases}$$

So this gives two numbers at a time. In practice, the subroutine is best written so that steps $1^\circ$ - $5^\circ$ are carried out every second step to get both $x$ and $y$. The first step returns $x$ and stores $y$. The second step only returns $y$ without any calculations at all.

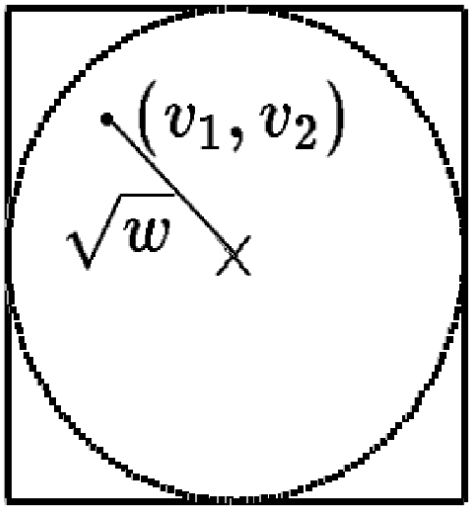
---

Algorithmically this method is fine, there is no wasted effort in terms of "misses" or something like that. But in terms of computational efficiency, it leaves much to be hoped for. It requires computation of the functions $\sqrt{}, \log, \sin$ and $\cos$. Calculating all of these, and especially the last three ones, is excruciatingly slow compared to the simple arithmetic operations. So it would be nice to have a more efficient routine.

---

It turns out that a variety of a hit-and-miss algorithm is actually almost always faster. The trick is to avoid having to calculate the sine and cosine explicitly.

Here we consider a simple unit circle:




and generate a point $(v_1, v_2)$ inside a unit circle centered at the origin:

**1**$^\circ$ Obtain $v_1 = P_u(-1, 1)$ and $v_2 = P_u(-1, 1)$ and $w = v_1^2 + v_2^2$

Then we check whether the point is really inside the circle:

**2**$^\circ$ If $w \geq 1$ return to step $1^\circ$

Then from basic geometry we know that the sine and cosine for the $\phi$ (which we do not know) of

the point $(v_1, v_2)$ are

$$\begin{cases} \cos\phi = v_1/\sqrt{w} \\ \sin\phi = v_2/\sqrt{w} \end{cases}$$

(Here we have to make sure $w$ can not be exactly 0!). Moreover, $w$ is of the form $P_u(0,1)$. Now we can again obtain the desired $x$ and $y$:

**3**° Calculate $r = \sqrt{-2\log w}$

**4**° Calculate

$$\begin{cases} x = r\cos\phi = rv_1/\sqrt{w} \\ y = r\sin\phi = rv_2/\sqrt{w} \end{cases}$$

The advantage here is that by using the latter parts of the equation above, we do not ever have to calculate the sine and cosine explicitly. This makes this approach faster, even though we do have to reject

$$\frac{4-\pi 1^2}{4} = 1 - \frac{\pi}{4} \approx 21\%$$

of all values in the hit-and-miss steps 1 and 2.

# 4.7. Generating random numbers on the surface of a sphere

[Own knowledge and derivation]

I will not discuss generating random numbers for multidimensional cases. The very basic cases are straightforward extensions of the 1D case: an $N$-dimensional uniform distribution is just a vector of $N$ 1-dimensional deviates, and the hit-and-miss approach works in any dimensions. If interested on generating Gaussian distributions in multidimensions, a good discussion can be found in the MC lectures of Karimäki (available on the course web page).

But there is one serious pitfall related to generating random deviates in 2 dimensions that it needs to be mentioned here separately. My and Kai Arstilas best guess is that this is probably one of the most common errors done by scientists in any kind of numerical analysis.

The problem is simply to select a random direction in 3 dimensions. That is, in spherical coordinates any vector can of course be given in the unit system $(r, \theta, \phi)$. To give a random direction, one then simply has to generate $\theta$ and $\phi$ randomly. So the obvious solution would seem to be to generate

$$\begin{cases} \theta = \pi P_u(0,1) \\ \phi = 2\pi P_u(0,1) \end{cases} \qquad (WRONG)$$

This is **outright wrong!**

---

Why is this? Consider the unit sphere $r = 1$. To generate random numbers on the surface of it (which is equivalent to generating a random direction), we want to have an equal number of points per surface area everywhere on the sphere. The surface area element $d\sigma$ is

$$\sin\theta d\theta d\phi$$

i.e. the area is not an even function of $\theta$. Hence one has to generate the $\theta$ random numbers distributed as

$$\sin\theta$$

rather than uniformly. Fortunately we know how to do this from section .

Let's for generality consider an arbitrary angular interval $[\alpha, \beta]$ lying inside the $[0, \pi]$ region. Then the normalization factor is

$$f_N = \int_\alpha^\beta \sin(\theta) d\theta = \cos\alpha - \cos\beta$$

and the normalized probability function is

$$f(\theta) = \frac{\sin\theta}{\cos\alpha - \cos\beta}$$

The cumulative function for an arbitrary interval $[\alpha, \beta]$ is

$$F(\theta) = \int_{\alpha}^{\theta} \frac{\sin\theta}{\cos\alpha - \cos\beta} d\theta = \frac{\cos\alpha - \cos\theta}{\cos\alpha - \cos\beta}$$

and the inverse can be solved as follows:

$$u = \frac{\cos\alpha - \cos\theta}{\cos\alpha - \cos\beta}$$

$$\Longrightarrow -\cos\theta = u(\cos\alpha - \cos\beta) - \cos\alpha$$

and hence

$$\theta = \cos^{-1}\left(\cos\alpha - u(\cos\alpha - \cos\beta)\right)$$

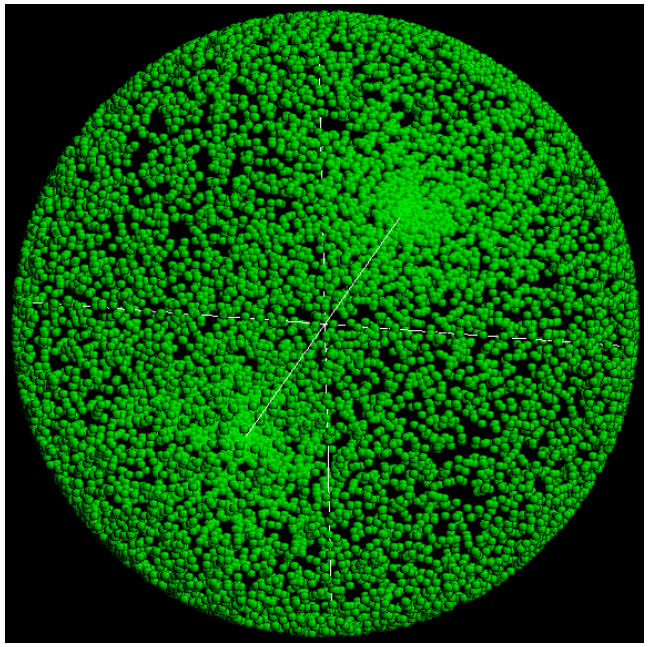
For the original case of a random direction anywhere in 3D, $\alpha = 0$ and $\beta = \pi$ and

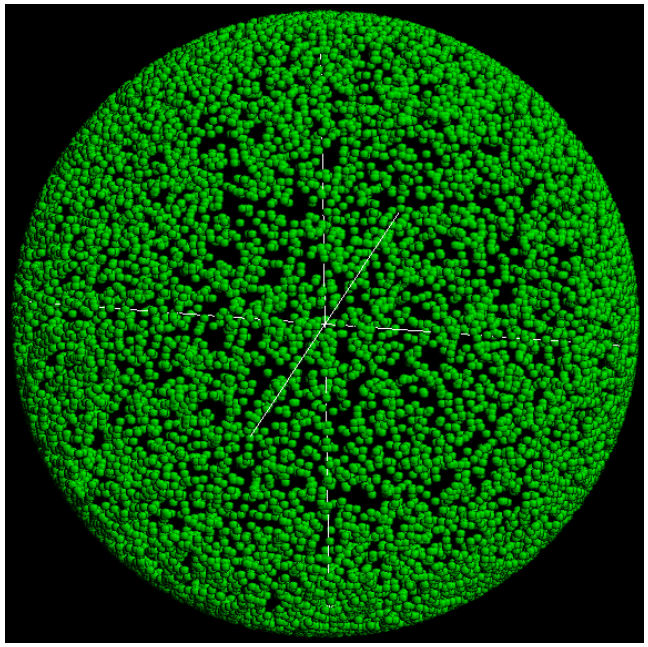
$$\theta = \cos^{-1}(1 - 2u)$$

which gives the correct algorithm

$$\begin{cases} \theta = \cos^{-1}\left(1 - 2P_u(0,1)\right) \\ \phi = 2\pi P_u(0,1) \end{cases}$$

If you do not believe me, compare the following two plots. They show 10000 points generated on the unit sphere, the left one using the wrong way to generate $\theta$, the right one using the correct way. The rotation is exactly the same in both figures, so they really are comparable.

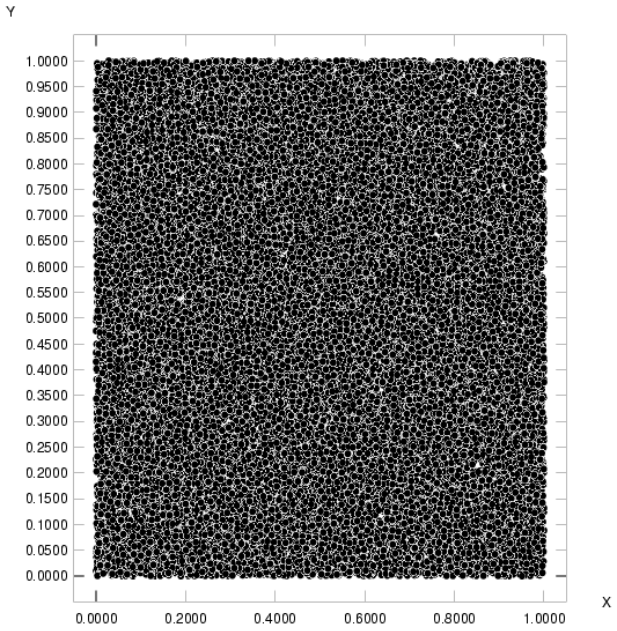


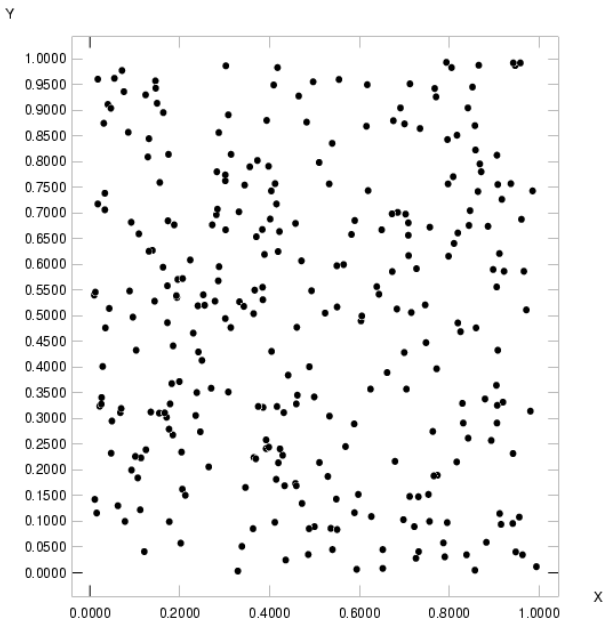


The problem is quite obvious in the left figure.

# 4.8. Non-random (quasi-) random distributions

Finally, we discuss pseudo-random numbers which are not random at all. Still, in some cases using these may be more efficient than using real pseudorandom numbers. This can be understood intuitively from the following two figures:




The left one shows 30000 random points in the unit box, the right one 300 points. From the left

side we see that there is nothing wrong with the distribution itself (on this scale at least): it fills the space. But the right side shows that with low statistics, there are clumps in the data here and there, and gaps elsewhere. This is as it should be for truly random numbers.

But this brings us to think that in case we want to integrate over the 2D plane, would it not be more efficient to arrange the numbers so that it is guaranteed that even for low statistics points lie fairly smoothly everywhere in the interval? This is the idea behind quasirandom numbers.

In fact, it has been shown that the convergence in MC integration behaves as follows:

$\propto \frac{1}{\sqrt{n}}$ for true and pseudo-random numbers.

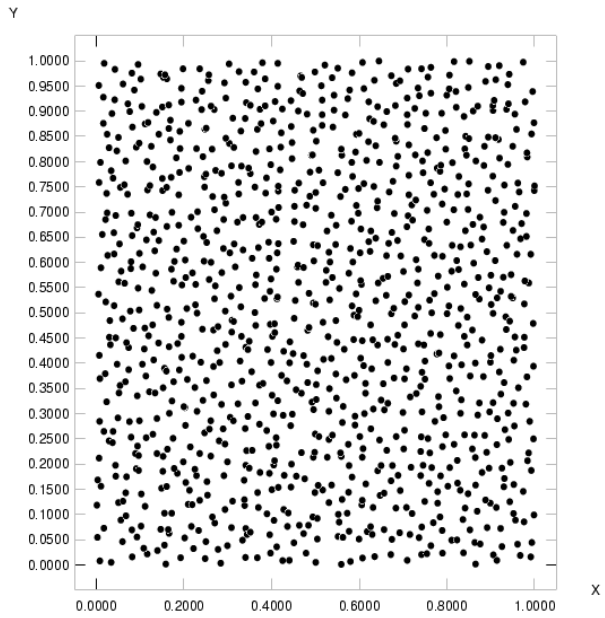
$\propto \frac{1}{n}$ at best for quasi-random numbers

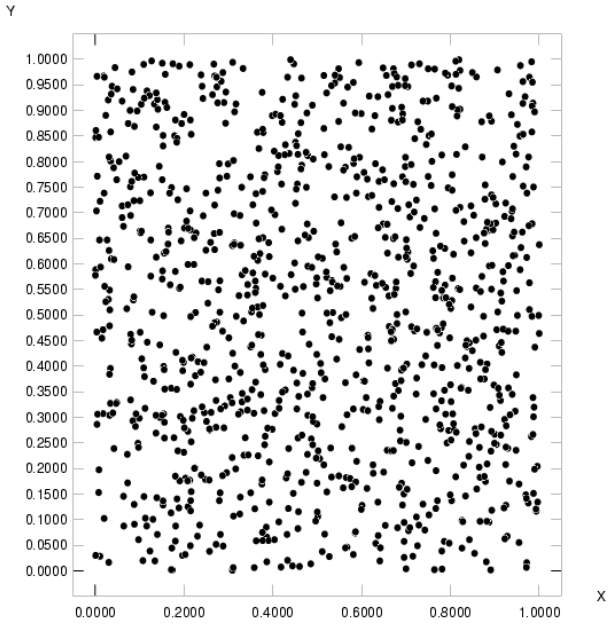
### 4.8.1. Stratified sampling

The simplest imaginable way to achieve this is to use **fully stratified** sampling. This means that we first decide how many points we want to generate, the divide our sample into exactly this many equal-sized boxes. Then we generate exactly one random number inside each box. This way it is guaranteed that the point distribution is fairly smooth.

This is illustrated in the following two figures: one has 1024 completely random numbers in 2D, the

other 1024 numbers generated with fully stratified MC using a $32 \times 32$ grid (you can figure out yourself which one is which):




Coding this is of course trivial, here is an example:

```
# Size of integration box
box=1.0;
# Interval size
gridsize=32;
```

```
for(i=0;i<gridsize;i++) {
    for(j=0;j<gridsize;j++) {
        x=(i+rand())*(box/gridsize);
        y=(j+rand())*(box/gridsize);
        print x,y;
        # Evaluate function to be integrated here
    }
}
```

This method has a significant practical drawback, however: one has to decide on the number of points needed in advance, and any intermediate result is worthless since parts of the 2D space have not been examined at all. One could introduce a mixing scheme which selects the minor boxes in random order to overcome the latter problem, but this would not solve the former.

---

A somewhat more flexible solution is to use **partially stratified** sampling. In here, we also divide the integration sample into boxes, but then select several points per box. This has the advantage that we can choose a somewhat smaller number of boxes, then do several loops where in every loop we select one point per box. This way we can stop the simulation anytime the outermost loop finishes.

This is probably clearer in code:

```
# Size of integration box
box=1.0;
# Number of intervals to do
ninterval=4;
# Interval size
gridsize=16;
for (interval=0;interval<ninterval;interval++) {
    for(i=0;i<gridsize;i++) {
        for(j=0;j<gridsize;j++) {
            x=(i+rand())*(box/gridsize);
            y=(j+rand())*(box/gridsize);
            print x,y;
            # Evaluate function to be integrated here
        }
    }
    # Intermediate result could be printed and simulation
    # stopped here
}
```

So we could in this example stop the run after 256, 512 or 768 steps in case we see we already have good enough statistics.

But even this is not very convenient. In large simulations, where the actual evaluation of a function can take hours or even days, the runs often die or have to be stopped in the middle, to be restarted later. For the stratified MC schemes one then would have to devise a restart which knows at which `i` and `j` values to restart. Quite possible to implement, but rather tedious.

## 4.8.2. Quasi-random numbers

[Karimäki notes p. 9, Numerical Recipes]

Fortunately it is quite possible to implement a number sequence such that it fills space evenly without having any specific "filling interval" like the stratified schemes. Such random number sequences are called **low-discrepancy numbers** or **quasi-random numbers.** Contrary to true random numbers, they are designed to be highly correlated, in a way such that they will fill space fairly evenly.

We give three examples of methods to generate such sequences.

### 4.8.2.1. Richtmeyer-sequences

We want to have a set of vectors $\mathbf{x}_i$ in $k$ dimensions

$$\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ik})$$

Each vector element is obtained as follows:

$$x_{ij} = i\sqrt{N_j} (\text{mod } 1)$$

where (mod 1) means we take the decimal part, and $N_j$ is the $j$:th prime number.

This is simple in concept, but not very efficient if large amounts of numbers are needed, since one also needs a list of prime numbers.

### 4.8.2.2. Van der Corput-sequences

This method is so simple that it can be easily explained, but in the default version it works only in 1D (you can think about yourself whether you could invent an extension to 2D).

The method is as follows:

1° Take an integer, and write it in some base (e.g. binary)

2° Reverse the digits in the number

3° Put a decimal point in front and interpret it as a decimal

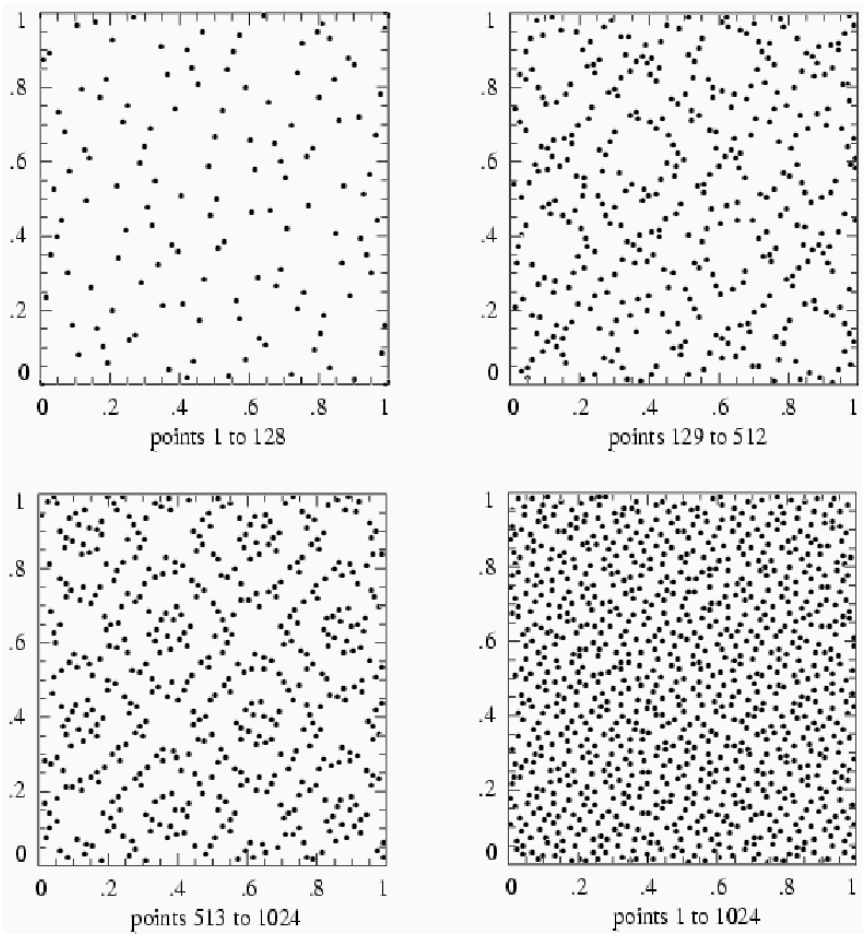
Using e.g. the binary system, we get:

| | | | | | |
|---|---|---|---|---|---|
| $i =$ | $\mathbf{1}$ | $1_2$ | $.1_2$ | $= \frac{1}{2}$ | $= 0.5$ |
| | $2$ | $10_2$ | $.01_2$ | $= \frac{1}{4}$ | $= 0.25$ |
| | $3$ | $11_2$ | $.11_2$ | $= \frac{1}{2} + \frac{1}{4}$ | $= 0.75$ |
| | $4$ | $100_2$ | $.001_2$ | $= \frac{1}{8}$ | $= 0.125$ |
| | $5$ | $101_2$ | $.101_2$ | $= \frac{1}{2} + \frac{1}{8}$ | $= 0.625$ |

### 4.8.2.3. Sobol sequences

The Sobol sequence is somewhat more complicated in definition, based on XOR operations and needing a list of initializing numbers. The code (and explanation of how it works) can be found in Numerical Recipes 2nd ed. chapter 7.7.

But here is an illustration of how it works:




We see that there are gaps, but these are always filled in on successive calling on the routine.

The Sobol sequence is not necessarily quite as efficient as an ideal quasi-random number generator.

The Sobol sequence efficiency for MC integration in $n$ dimensions is

$$O((\ln N)^n/N$$

whereas the optimal efficiency is $1/N$. Still, this is not bad at all compared to the $1/\sqrt{N}$ efficiency obtained by basic pseudo-random numbers.

To summarize, if you think quasi-random numbers might work better than pseudorandom numbers in your application, what should you do? First check the literature on whether someone has examined this in a problem similar to yours. If not, simply test it. I personally would first go for partially stratified MC or a Sobol sequence (since the code is readily available). And if the number of dimensions is low ($< 6$), I would also test a regular grid (no randomness at all!).

# 4.9. Final warning

In this section, I have not really been too exact about the question of whether the limits, $a$ and $b$, are allowed values when returning random numbers in the range $[a, b]$. This is because mathematically it does not matter: the probability of hitting exactly the limit is of course infinitely small.

---

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. Since a present-day simulation can easily go through 10 million numbers in a fraction of a second, this is actually a quite large probability. Even for 8-byte floating point variables, with a 15-digit mantissa, the probability of hitting the limit is not negligible at all. Hence working with any random number generator, you have to consider whether the generator can return the end points (0 and 1 for $P_u(0, 1)$ ) and make sure this is compatible with your simulation.

Some of the routines given above would actually die immediately in a division by zero in case the generator can return exactly 0.0.

# 4.10. Final remark

Back in the 1940's, when the very first generators were designed, von Neumann thought that:

> *Anyone who considers arithmetic methods of*
> *producing random digits*
> *is, of course, in a state if sin.*
> J. von Neumann

---

Now, when generators tend to be at least fairly decent and are very widely used, this fortunately is no longer true!