Advanced Programming for Scientific Computing (PACS)
Lecture title: Random numbers and distributions

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

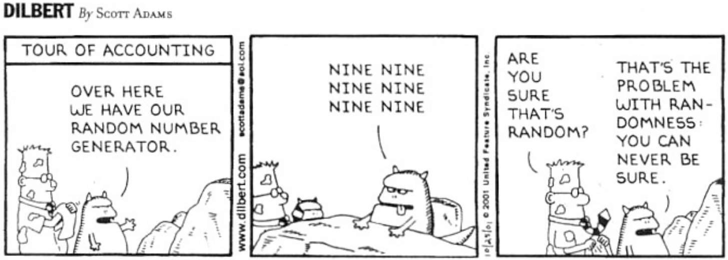
A.A. 2021/2021

Random numbers are important!

The capability of generating random number is essential not only for statistical purposes but also for internet communications. But an algorithm is deterministic. However, several techniques have been developed to generate pseudo-random numbers. They are not really random, but they show a low level of auto-correlation.

Is a sequence of numbers not random?

Well, you can never be sure!


DILBERT By Scott Adams
TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
www.dilbert.com
scottadams@aol.com
© 2001 United Feature Syndicate, Inc.

(Taken from igmur.com)

But you can look at the autocorrelation properties of the sequence (see this wikipedia page), which however gives you only the probability that a sequence is not-random, you can never be sure!

LavaRand

Generating real random number is so important (particularly for cryptography) that fancy techniques have been invented to have a large amount of entropy.

Did you now that 10% of internet security is guaranteed by a random number generator based on a wall of lava lamps? The chaotic evolution of the two fluids inside the lamp is monitored and used to generate random numbers.

Have a look at this video on YouTube

C++ support for statistical distributions

C++ provides an extensive support for (pseudo) random number generators and univariate statistical distribution. You need the header `<random>`. The chosen design is based on two types of objects:

- **Engines** serve as a *stateful source of randomness* They are function objects that provide random unsigned integer values uniformily distributed in a range. They are normally used with Distributions;
- **Distributions**. They specify how values generated by the engine have to be transformed to *generate a sequence with prescribed statistical properties*.

The design separates the *(pseudo) random number* generators, implemented in the engine, from their use to generate a specific distribution.

Engines

Random number engines generate pseudo-random numbers using seed data as entropy source. Several different classes of pseudo-random number generation algorithms are implemented as templates that can be customized. We have a set of basic engines that implement well known pseudo-random generators. They are class templates (I do not indicate here the template parameters for brevity)

linear_congruential_engine<>	linear congruential algorithm
mersenne_twister_engine<>	Mersenne twister algorithm
subtract_with_carry_engine<>	subtract-with-carry-algorithm (a lagged Fibonacci)

However, for simplicity of use, the library provides some *predefined engines*, which are in fact instances of the above with predefined values of the template argument.

Predefined engines

The main one is default_random_engine, which should guarantee a good compromise between efficiency and quality of the (pseudo) random numbers.
Others are minstd_rand0, minstd_rand, mt19937 mt19937_64, ranlux24_base,ranlux48_base, ranlux24, ranlux48 and knuth_b.

We have also a non deterministic random number generator, called random_device, in case you hardware provides a non-deterministic random number generator (all other engines are deterministic, thus pseudo-random generators)

Details may be found in in this web page.

The random_device

As said, random_device is a particular random number generator engine which creates the random numbers by elaborating data of your computer hardware (as disk speed, CPU temperature, n. of processes currently running...), in an attempt to produce real random numbers.

However, it is rather costly. Often, is only used to generate the seed for another random engine.

How to use a predefined engine

It is very simple. You generate an object of the chosen class either with the default constructor or providing a seed (a part random_device, which does not require a seed).

```
std::default_random_engine rd1;
// with a seed
std::default_random_engine rd2{1566770};
```

Seeds are unsigned integers (in fact 32 bits unsigned) that you use to introduce entropy. If you use the same seed the sequence of pseudo random number will be the same every time you execute the program. (Note: the default_random_engine guarantees this property only on the same architecture).

Random engines should never be used alone, but always together with distributions

Distributions

Distributions are template classes (I omit to indicate the template parameters because the default values are usually ok). They implement a call operator () that accepts in input an engine object: they transform the random sequence into the wanted distribution.

```
#include <random>
#include <iostream>
int main(){
  std::default_random_engine gen;
  std::uniform_int_distribution<> dice{1, 6};
  for(int n=0; n<10; ++n)std::cout << dice(gen) << ' ';
  std::cout << std::endl;
}
```

Here `uniform_int_distribution<> dis(1, 6)` provides an integer uniform distribution in the range $(1, 6)$.

Distributions

The arguments of the constructor of a distribution (if it is implemented as a class) vary depending on the distribution. However, all distributions have default values. The list of distributions available is very long. We list only some:

Uniform	uniform_int_distribution, uniform_real_distributions
Bernoulli	bernoulli_distribution, binomial_distribution
Poisson	poisson_distribution, exponential_distribution, gamma_distribution
Normal	normal_distribution, lognormal_distribution,cauchy_distribution
Sampling	discrete_distribution, piecewise_linear_distribution

A full list in this web page

How to use the random_device

The random_device provides a non-deterministic random number generator. However, it is much slower than the other engines. Therefore, it is normally used to generate the seed for another (pseudo) random engine.

Here the Knuth engine is initialized with a seed generated by the random device:

```
std::random_device rd;
std::knuth_b reng{rd()};
...
```

An utility: seed_seq

std::seed_seq consumes a sequence of integer-valued data and produces a requested number of unsigned integer values i, $0 \leq i < 2^{32}$, based on the consumed data. The produced values are distributed over the entire 32-bit range even if the consumed values are close.

It provides a way to seed a large number of random number engines or to seed a generator that requires a lot of entropy, given a small seed or a poorly distributed initial seed sequence.

```
std::seed_seq seq={1,2,3,4,5};
std::vector<std::uint32_t> seeds(10);// vector of 10 elmts.
seq.generate(seeds.begin(), seeds.end());
```

Now you may use seeds to feed different random engines. See the possible output here.

Shuffling

In the header <algorithm> you have the utility shuffle that shuffles a range of elements so that each possible permutation has the same probability of appearance. Here an example of usage

```
std::vector<int> v={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::random_device rd;
std::default_random_engine g{rd()};
std::shuffle(v.begin(), v.end(), g);
// v has been shuffled
```

Every time I run this piece of code vector v is permuted in a different way.

Sampling

Another interesting utility in <algorithm> is sample, which extracts from a range n elements (without repetition) and inserts them into another range.

```
int n =10;
std::vector<double> p;
// fill p with m>n values to sample
std::vector<double> res;
std::sample(p.begin(),p.end(),std::back_inserter(res),
n, std::mt19937{ std::random_device{}() } )
```

Here, I am using the std:mt19937 engine, seeded by a random_device constructed on the fly.

Every time I run this code I generate a different realization of the sample.

Examples

In the directory
RandomDistributions
you find a simple example of various distributions.