4-random-numbers

June 3, 2024

# Pseudo-random number generation

As compared to C, the C ++ standard library takes a big step forward in providing multiple utilities for generating pseudo-random numbers with different properties and probability laws.

One of the dilemmas of a developer of scientific code is to choose between a non-deterministic mechanism, therefore closer to what one expects from the "random" notion, and a deterministic and reproducible mechanism.

0.1 Within C

The generation of random numbers within C relies on the `rand()` function, which returns a pseudo-random integer between `0` and `RAND_MAX`.

The srand() function allows one to set the starting point of the process, the seed. We can give it a fixed value if we want to reproduce the same sequence each time and to always obtain the same final result.

[1]:
```
#include <iostream>
#include <cassert>   // for assert
#include <cstdlib>   // for rand
#include <ctime>     // for time
#include <array>     // for std::array
#include <numeric>   // for accumulate
```

[2]:
```
constexpr unsigned SIZE = 1024 ;
std::array<double,SIZE> coll ;
```

[11]:
```
std::srand(1) ;
for ( auto & elem : coll )
 { elem = std::rand()/(RAND_MAX+1.)-0.5 ; }

double mean = 0. ;
for ( auto elem : coll )
 { mean += elem ; }
mean /= SIZE ;

std::cout<<"mean   : "<<mean<<std::endl ;
```

```
mean   : 0.00675054
```

In order to be **more random (but less reproducible)**, it is common to mix a call to `srand ()` with a call to `time(0)`.

[8]:
```
std::srand(std::time(0)) ;
for ( auto & elem : coll )
 { elem = std::rand()/(RAND_MAX+1.)-0.5 ; }

double mean = 0. ;
for ( auto elem : coll )
 { mean += elem ; }
mean /= SIZE ;

std::cout<<"mean   : "<<mean<<std::endl ;
```

```
mean   : -0.00338041
```

0.2 C++ engines

Instead of the single `rand()` function, the C++11 standard library provides a set of engines named after the algorithm which is used: * `linear_congruential_engine` * `mersenne_twister_engine` * `subtract_with_carry_engine` (also called Lagged Fibonacci)

All those engines are **function objects**: we start by calling a constructor of the class, optionally giving a seed, then we can use the created object as if it were a function, to produce a pseudo-random number given by its `operator()`.

[12]:
```
#include <iostream>
#include <random>
```

[13]:
```
std::minstd_rand0 engine ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
```

```
16807
282475249
1622650073
984943658
1144108930
```

The standard library also defines a `default_random_engine`, whose type is implementation dependent.

[21]:
```
std::default_random_engine engine ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
```

```
std::cout<<engine()<<std::endl ;
```

```
16807
282475249
1622650073
984943658
1144108930
```

One of the difficulties with those engines is that they are all different C++ types. Even worse, the type is unknown for default engine. If you want some flexibility for the choice of the engine, you may have no choice but to use template parameters.

If one wants truly random numbers, instead of `time()`, C++11 provides **`std::random_device`**, which is supposed to rely on "a source of hardware entropy", *if the hardware permits*, in order to provide non-deterministic random integers. Same as engines, it is an object-fonction which must be constructed first, then used as a function so to produce the integer than can be used as a seed for another engine.

[18]:
```
std::random_device device ;
std::default_random_engine engine(device()) ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
```

```
784308858
624351120
864174598
750563925
408944997
```

Note: the seed can also by set after the construction of a generator, by calling the member function `seed()`.

[16]:
```
std::random_device device ;
std::default_random_engine engine ;
engine.seed(device()) ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
std::cout<<engine()<<std::endl ;
```

```
1175761748
2030662589
1536015199
906529006
1784012024
```

0.3 Distributions

On top of the engines, the library provides a set of **distributions**. This makes it possible to transform the (pseudo-)random integers supplied by engines into a particular probability law: uniform, bernouilli, geometric, Poisson, binomial, uniform, exponential, normal and gamma.

BEWARE: distribution are object-functions, which must be constructed first, and then take an engine as input argument later, when they are called as functions.

Below, we generate a normal (Gaussian) distribution with its default parameters and verify that the mean tends to `0` and the standard deviation to`1`. As with engines, distributions are function objects. Their `()` operators take a generator as argument.

[17]:
```
#include <iostream>
#include <array>
#include <random>
#include <cmath>

constexpr unsigned SIZE = 1024 ;
std::array<double,SIZE> coll ;
```

[18]:
```
std::default_random_engine engine ;
std::normal_distribution<double> distrib{0.,1.} ;
for ( auto & elem : coll )
 { elem = distrib(engine) ; }

double mean = 0. ;
for ( auto elem : coll )
 { mean += elem ; }
mean /= SIZE ;

double stddev = 0. ;
for ( auto elem : coll )
 { stddev += std::pow(elem-mean,2.) ; }
stddev = std::sqrt(stddev/SIZE) ;

std::cout<<"mean   : "<<mean<<std::endl ;
std::cout<<"stddev : "<<stddev<<std::endl ;
```

```
mean   : -0.0186748
stddev : 0.994092
```

1 Questions?

2 Exercise

1. In the example below there is an error which causes the final result to be always be 0. Will you find out why?
2. Try to run the program with the deterministic and non-deterministic arguments.

3. Modify the program so to use the C++ standard library.

BEWARE: if you work with CoLiRu, `std::random_device` is not really random.

[22]:
```
%%file tmp.random-numbers.cpp

#include <iostream>
#include <cassert>   // for assert
#include <cstdlib>   // for rand
#include <ctime>     // for time
#include <array>     // for std::array
#include <numeric>   // for accumulate

int main( int argc, char * argv[] )
 {
  assert(argc==2) ;
  std::string mode(argv[1]) ;

  if (mode=="deterministic")
   { srand(1) ; }
  else if (mode=="non-deterministic")
   { srand(std::time(0)) ; }
  else throw "unknown mode" ;

  constexpr unsigned SIZE = 1024 ;
  std::array<double,SIZE> coll ;
  for ( double & elem : coll )
   { elem = std::rand()/(RAND_MAX+1.)-0.5 ; }


  double mean = std::accumulate(coll.begin(),coll.end(),0)/SIZE ;
  std::cout<<mean<<std::endl ;
 }
```

```
Overwriting tmp.random-numbers.cpp
```

[23]:
```
!rm -f tmp.random-numbers.exe && g++ -std=c++17 tmp.random-numbers.cpp -o tmp.
↪random-numbers.exe
```

[24]:
```
!./tmp.random-numbers.exe deterministic
```

```
0
```

3 Resources

- cppreference