Ranvec1
Random number generator for C++ vector class library

Agner Fog

Libre Planet

Free Software

Chapter 1

Introduction

Ranvec1 is an efficient high quality pseudo random number generator designed for large vector applications and multi-threaded applications in C++ language.

This generator has been developed based on the following design goals:

- Good randomness, as determined by both theoretical and experimental criteria.
- Suitable for vector processors and vector instructions (SIMD).
- Suitable for large multi-threaded applications without risk of overlapping subsequences.
- Fast generation of large amounts of random numbers.

This random number generator is designed for large Monte Carlo simulations and Monte Carlo integration. It may be useful for cryptographic applications as well, but cryptographic safety has not been a decisive design goal. It will be useful for game applications as well.

A physical random number generator function is included for the purpose of generating a truly random seed for initializing the pseudo random number generator.

The code is based on the Vector Class Library, using the x86 or x86-64 instruction set with extensions from SSE2 to AVX512. See the Vector Class Library manual for choice of compiler and compilation options. On Gnu and Clang compilers you need to specify the additional options **-mrdrnd -mrdseed** in order to enable the physical random number generator instructions.

Chapter 2

Instructions

The files ranvec1.h and ranvec1.cpp define a high quality pseudo-random number generator with vector output. This generator is useful for producing random numbers for simulation and other Monte Carlo applications. Add the file ranvec1.cpp to your project and compile for the appropriate instruction set. This example shows a simple use of the random number generator:

Example 2.1.

```
// Example for random number generator
// Remember to link ranvec1.cpp into the project

#include <stdio.h>
#include "vectorclass.h"
#include "ranvec1.h"

int main() {
    // Arbitrary seed
    int seed = 1;
    // Create an instance of Ranvec1 and set the type to 3
    Ranvec1 ran(3);
    // Initialize with the seed
    ran.init(seed);
    // Generate a vector of 8 random integers below 100
    Vec8i ri = ran.random8i(0,99);
    // Generate a vector of 8 random floats
    Vec8f rf = ran.random8f();
    int i;
    // Output the 8 random integers
    printf("\nRandom integers in interval 0 - 99\n");
    for (i=0; i < ri.size(); i++) printf("%3i ", ri[i]);

    // Output the 8 random floats
    printf("\nRandom floats in interval 0 - 1\n");
    for (i=0; i < rf.size(); i++) printf("%7.4f ", rf[i]);
    printf("\n");
    return 0;
}
```

The optional parameter for the constructor of the class Ranvec1 defines the type of random number generator to use:

Parameter for constructor	Generator type
1	MWC. Multiply-With-Carry Generator. Use this for small applications where speed is important. (cycle length $> 4 \cdot 10^{19}$)
2	MTGP. A variant of Mersenne Twister. Use this for applications with multiple threads. (cycle length $> 10^{3375}$)
3	MWC + MTGP combined. Use this for the best possible randomness and for large applications with many threads. (cycle length $> 10^{3395}$)

It is necessary to initialize the random number generator with a seed, using either the function `init` or `initByArray`. The generator will produce only zeroes if it has not been initialized with any of the init functions.

The random number sequence depends on the seed. A different seed will produce a different sequence of random numbers. You can reproduce a random number sequence exactly after initializing again with the same seed. You may use simple values like 1, 2, 3, ... for seeds in a series of simulations if you want to be able to reproduce the results later. If you want a non-reproducible sequence then you need a seed from a source of genuine randomness. The function `physicalSeed` is useful for this purpose.

The generator can produce vector outputs with different vector sizes. The best performance is obtained when the vector size fits the instruction set: SSE2 or higher for 128 bit vectors. AVX2 or higher for 256 bit vectors. AVX512 or higher for 512 bit vectors. Depending on details of the application, it may or may not be possible to reproduce a simulation result exactly when the vector size is changed.

The theory of the Ranvec1 package including the different generators, multiprocessing and vector processing is described in the article:
Fog, Agner: "Pseudo-Random Number Generators for Vector Processors and Multicore Processors." Journal of Modern Applied Statistical Methods, vol. 14, no. 1, 2015, article 23.
`https://digitalcommons.wayne.edu/jmasm/vol14/iss1/23/`

2.1 Member functions for class Ranvec1

Constructor	Ranvec1(int gtype)
Description	Constructor for Ranvec1 class. See the table above for values of the generator type gtype.
Efficiency	medium

```
// Example:
Ranvec1 ran(3);   // Create object ran
```

Member function	void init(int seed)
Description	Initialization with one seed. Any value is allowed for seed. Use a different value of seed each time to get a different random number sequence.
Efficiency	poor

```
// Example:
ran.init(0);   // Initialize random generator with seed 0
```

Member function	void init(int seed1, int seed2)
Description	Initialization with two seeds. The random number sequence depends on both seeds. If the generator type is 3, then seed1 is used for the MWC generator and seed2 is used for the MTGP generator. The value of seed2 should be different for each thread in multithreaded applications.
Efficiency	poor

```
// Example:
ran.init(0,1);   // Initialize random generator with seeds 0 and 1
```

Member function	void initByArray(int const seeds[], int numSeeds)
Description	Initialization with multiple seeds. The seeds array must contain numSeed integers. The random number sequence depends on all these integer seeds. This can be useful for security applications in order to make it difficult to guess the seeds. The best security is obtained with generator type 3.
Efficiency	poor

```
// Example:
// Initialize random generator with four seeds
int seeds[4] = {5,8,12,2};
ran.initByArray(seeds, 4);
```

Member function	uint32_t random32b() uint64_t random64b()
Description	returns an integer of 32 or 64 random bits
Efficiency	medium

```
// Example:
unsigned int r = ran.random32b();  // generate 32 random bits
```

Member function	Vec4ui random128b() Vec8ui random256b() Vec16ui random512b()
Description	Returns an integer vector of 128, 256 or 512 random bits.
Efficiency	medium

```
// Example:
Vec8ui v = ran.random256b();  // generate 256 random bits
```

Member function	int random1i(int min, int max) Vec4i random4i(int min, int max) Vec8i random8i(int min, int max) Vec16i random16i(int min, int max)
Description	Returns a random integer or a vector of random integers with uniform distribution in the interval min $\leq$ x $\leq$ max. (The distribution may be slightly inaccurate when the interval size is large and not a power of 2. See below for a more accurate version.)
Efficiency	medium

```
// Example:
```

```
// Generate a random integer in the interval [1,10]
int r = ran.random1i(1, 10);
// Generate eight random integers in the interval [1,10]
Vec8i v = ran.random8i(1, 10);
```

Member function	int random1ix(int min, int max) Vec4i random4ix(int min, int max) Vec8i random8ix(int min, int max) Vec16i random16ix(int min, int max)
Description	Returns a random integer or a vector of random integers with uniform distribution in the interval min $\leq$ x $\leq$ max. This is the same as random1i, random4i, random8i, random16i, but exact. The exact version of these functions use a rejection method as described in the theory article mentioned above. To reproduce a sequence, the same function with the same vector size must be called.
Efficiency	medium

```
// Example:
// Generate eight random integers in the interval [1,10]
Vec8i v = ran.random8ix(1, 10);
```

Member function	float random1f()
Description	Returns a random floating point number with uniform distribution in the interval 0 $\leq$ x $<$ 1. The resolution is 2^{-24}. (A value in the interval 0 $<$ x $\leq$ 1 can be obtained as 1 - x.
Efficiency	medium

```
// Example:
// Generate a random float below 100:
float x = ran.random1f() * 100.f;
```

Member function	Vec4f random4f() Vec8f random8f() Vec16f random16f()
Description	Returns a vector of random floating point numbers with uniform distribution in the interval 0 $\leq$ x $<$ 1. The resolution is 2^{-24}.
Efficiency	medium

```
// Example:
// Generate four random float numbers below 100:
Vec4f v = ran.random4f() * 100.f;
```

Member function	double random1d()
Description	Returns a random double precision number with uniform distribution in the interval 0 $\leq$ x $<$ 1. The resolution is 2^{-52}.
Efficiency	medium

```
// Example:
// Generate random double precision number below 100:
double x = ran.random1d() * 100.;
```

Member function	Vec2d random2d() Vec4d random4d() Vec8d random8d()
Description	Returns a vector of random double precision numbers with uniform distribution in the interval $0 \leq x < 1$. The resolution is 2^{-52}.
Efficiency	medium

```
// Example:
// Generate four random double precision numbers below 100:
Vec4d v = ran.random4d() * 100.;
```

2.2 Other functions

Function	int physicalSeedType()
Description	Finds the best source of non-reproducible randomness on the CPU that the program is running on. Return value: 0: No physical seed available 1: CPU clock (consecutive calls are not independent) 2: RDRAND instruction 3: RDSEED instruction
Source file	physseed.cpp
Efficiency	medium

Function	int physicalSeed()
Description	Get a non-reproducible random number based on a physical process. This is intended as a seed for the pseudo random number generator. The source of randomness is indicated by physicalSeedType();
Source file	physseed.cpp
Efficiency	medium

```
// Example: Generate a random seed
int seed = physicalSeed();
// Make an instance of the pseudo random number generator
Ranvec1 ran(2);
// Initialize it with the random seed
ran.init(seed);
// Generate a vector of 16 random float numbers
Vec16f rf = ran.random16f();
// This code will generate a different random sequence each
// time it runs.
```

2.3 Generating seeds

Ranvec1 is called a pseudo random number generator because it is deterministic. You can repeat the same sequence of random numbers if you run it again with the same seed. You need to initialize Ranvec1 with a random seed if you want a sequence of random numbers that is not predictable or deterministic.

The `physicalSeed()` function will produce such a random seed. Newer CPUs have a built-in physical source of randomness based on thermal noise. This is implemented in the RDRAND or RDSEED instruction. The RDSEED instruction is stronger than RDRAND if you want to call it multiple times to get a longer seed. The `physicalSeed()` function will use the best source of randomness available on the CPU it is running on.

If the program is running on and older CPU without the RDRAND or RDSEED instruction, then you can use the internal CPU clock as a source of randomness. The frequency of this internal clock is typically higher than 1 GHz. The source of randomness here is the exact time at which the function is called.

Note that if you are calling `physicalSeed()` twice on an older computer where the CPU clock is the only source of randomness, then the second call will not be independent of the first one. It will give a value that is perhaps a few hundred clock counts higher than the first one. To get an independent second value you need to wait for some external event before the second call. This external event can be a keystroke, a mouse move, or a network event. If the function `physicalSeedType()` returns 1 then you need to wait for an external event before every call to `physicalSeed()` except the first one. For example, you may ask the user to press a key.

2.4 Cryptographic applications

It is theoretically possible to predict and reproduce the sequence generated by a single pseudo random number generator if you have access to a subsequence longer than the internal state buffer. This is not possible if two random number generators with long cycle lengths are combined. Therefore, you should always use the combined generator (type 3) for cryptographic applications.

You should use a seed longer than 32 bits to get a good unpredictable result. Use the `initByArray` function with an array of multiple seeds. Use two or more array elements generated by the `physicalSeed()` function and supply with other elements from other sources. These other elements do not need to be truly random; they may include date and time, a hash of the user name or password, or any other data. The resulting random number sequence depends on all the elements in the seeds array. The resulting sequence will be unpredictable as long as at least one element of the seeds array is truly unpredictable. Combining seeds from multiple sources makes it more difficult for an attacker to break the security.

2.5 Game applications

The source of randomness does not need to be highly secure for entertainment games. A single seed from the `physicalSeed()` function will provide sufficient randomness.

2.6 Gambling applications

Gambling is a morally dubious exploitation of well-known weaknesses in the human psyche for financial gain, in my opinion. I do not endorse the use of this software in gambling applications.

2.7 Monte Carlo simulation

Monte Carlo simulation and Monte Carlo integration are computational techniques that require a very long sequence of random numbers. The Ranvec1 generator was designed to be well suited for this purpose.

You do not need truly unpredictable randomness for Monte Carlo applications. On the contrary, it is an advantage to have a deterministic sequence so that it is possible to re-play a particular simulation in case of an interesting event that you want to analyze further. It is quite convenient to use consecutive seeds such as 1, 2, 3, ... for a series of simulation runs.

2.8 Multi-threaded applications

The Ranvec1 generator is designed to be suitable for large multi-threaded applications. You can take advantage of the multiple CPU cores in modern computers by running multiple threads simultaneously in time-consuming applications. The number of threads should not be more than the number of CPU cores. Some microprocessors are able to run two or more threads in each core. In this case, the number of logical processors is higher than the number of physical processors. Two threads running in the same core are likely to be competing for the same resources, so it may not be efficient to run more threads than CPU cores in this case.

It is not safe to access a pseudo random number generator from multiple threads simultaneously. Instead, you need to make one instance of Ranvec1 for each thread. Each instance should have a different seed. It is recommended to use the combined generator (type 3) with two seeds. The second seed, or both seeds, should be different for each thread. The theoretical reasons for this are explained in the theory article cited on page 3.

Example 2.2 shows how to generate random numbers in multiple threads. Note that there will be one instance of the random number generator object `Ranvec1` in each thread because it is declared inside the thread function.

Example 2.2.

```
// Example of random number generation with multiple threads
// random_threads.cpp

// Example of command line options for g++ and clang:
// g++     -O2 -std=c++17 -mavx2 -mfma -pthread random_threads.cpp
// clang++ -O2 -std=c++17 -mavx2 -mfma -pthread random_threads.cpp

// for Visual Studio only: define desired instruction set:
// #define INSTRSET 8

#include <stdio.h>
#include <thread>

#include "ranvec1.h"    // random number generator
#include "ranvec1.cpp"  // put code in separate module or include

// Thread function. Will run one instance for each thread
// This function calculates the mean of 1000 random numbers
void thread_function(int threadnum, int seed, double * result) {

    // Make an instance of the random number generator
    // (this instance is local to each thread)
    Ranvec1 ran(3);

    // Initialize. Use the thread number as a second seed to get
    // different results in each thread
    ran.init(seed, threadnum);

    // Accumulator for eight sums
    Vec8d accum = 0.;

    // Generate 1000 random double precision numbers
    for (int i = 0; i < 125; i++) {
        // Vector of eight double precision random numbers
        accum += ran.random8d();
    }
    // Calculate sum and mean
    double sum = horizontal_add(accum);
    double mean = sum * 0.001;

    // Return result
    *result = mean;
}

int main() {

    // Number of threads
    const int number_of_threads = 4;

    // Array of thread objects
    std::thread threads[number_of_threads];

    // Array of results
    double results[number_of_threads];

    // Arbitrary seed
    int seed = 25;

    // Start threads
    for (int t = 0; t < number_of_threads; t++) {
        threads[t] =
        std::thread(thread_function, t, seed, &results[t]);
    }

    // Wait for threads to finish
    for (int t = 0; t < number_of_threads; t++) {
        threads[t].join();
    }

    // write results
    for (int i = 0; i < number_of_threads; i++) {
        printf("%.6f  ", results[i]);
    }

    return 0;
}
```