Variate Generator Library

Feng Wang
feng.wang@uni-ulm.de

March 6, 2014

Contents

1 Getting Started

1.1 Welcome

Welcome to Variate Generator library!

By the time you have read through this tutorial, you will be able to play with it.

1.2 Compilation Enviroment Requirement

As some C++11* features are employed when implementing this library, before we get further, please check your compiler for C++11 compatablity.

1.3 A Quick Example

The code listed in Table 1 shows how to generate gaussian random numbers:

Table 1: Source Code for a Gaussian Variate Example

```
#include <vg.hpp>
#include <cmath>
#include <map>
#include <iostream>
int main()
{
    //generate double precision gaussian random numbers
    //using mt19937 as random generator engine with arguments (0,4)
    vg::variate_generator<double, vg::gaussian, vg::mt19937> vg(0, 4);
    std::map< int, int > sample;
    //generate 500 gaussian numbers and store them in a map
    for ( auto i = vg.begin(); i != vg.begin()+500; ++i )
```

*Features such as lambda functions, variadic template and keyword auto, see http://www.open-std.org/jtc1/sc22/wg21/ for more information.

```
        sample[std::round(*i)]++;
    //show the number generated
    for ( auto i = sample.begin(); i != sample.end(); ++i )
    {
        std::cout << (*i).first << "\t";
        for ( auto j = 0; j < (*i).second; ++j )
            std::cout << "*";
        std::cout << "\n";
    }
    return 0;
}
```

As this library is header file only, if your c++ compiler is g++, a typical compilation and link command for the example code whose file name is *test_gaussian.cc* can be

g++ -I*PATH_TO_THE_HEADER* -o ./bin/gaussian_test test_gaussian.cc -std=c++11

This command will generate a executable file *gaussian_test* in directory *./bin*, and its executation result is shown in figure 1

2 An Overview of VG

A random variate generator consists of three parts:

- variate type
- distribution type
- engine type

2.1 struct variate_generator

The basic struct used for a generator is variate_generator, which is decleared as:

Table 2: variate_generator struct

```
namespace vg
{
    template
    <
     class T = double,                                    //variate
     template<class, class> class Distribution = uniform,//distribution
     class Engine = mitchell_moore                       //engine
    >
    struct variate_generator
    {
       template< typename ... Tn >
       variate_generator( const Tn ... );
       T operator()() const;
       operator T() const;
       iterator begin() const;
    };
};
```

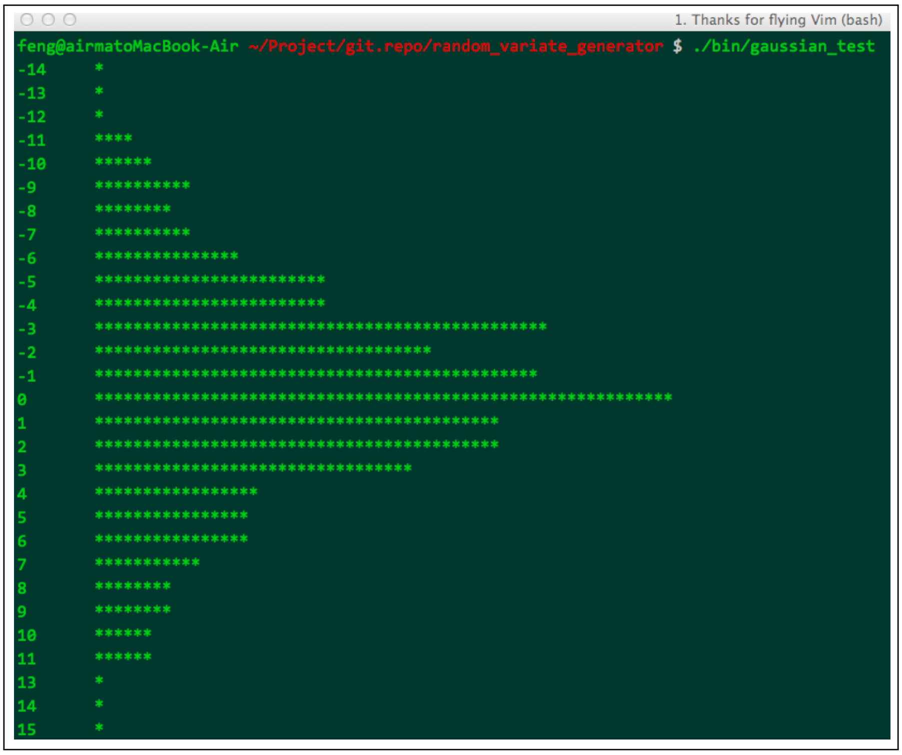

Figure 1: Gaussian Random Number Example

2.1.1 declaration

So to make a generator to product variates of int type and lagarithmic distribution, with a parameter 0.33, we can simply declare:

```
vg::variate_generator<int, vg::lagarithmic> v( 0.33 );
```

which it is equivalent to

```
vg::variate_generator<int, vg::lagarithmic, vg::mitchell_moore> v( 0.33, 0 );
```

where the last argument 0 is the default engine seed.

Also, to make a generator to product variates of hypergeometric distribution, with int type and paramenters (200, 200, 200), using mt19937 persudo–random engine and engine seed 987654321, we can declare it with one line code like this:

```
vg::variate_generator<int, vg::hypergeometric, vg::mt19937> v( 200, 200, 200, 987654321 );
```

2.1.2 Generation

After the generator *v* has been declared, we can generate variate in several ways:

Generate only one variate:

```
auto i = v();
int j = v;
auto k = *(v.begin());
```

Generate multiple variates:

```
std::vector<int> array1( v.begin(), v.begin()+100);
```

```
std::vector<int> array2( 100 );
std::generate( array2.begin(), array2.end(), v );
std::vector<int> array3;
std::copy( v.begin(), v.begin()+100, std::back_inserter( array3 ) );
```

2.2 Built-in Distributions

Curent we have about fifty distributions implemented:

- arcsine distribution
- bernoulli distribution
- beta distribution
- beta_binomial distribution
- beta_pascal distribution
- binomial distribution
- burr distribution
- cauchy distribution
- chi_square distribution
- digamma distribution
- erlang distribution
- exponential distribution
- exponential_power distribution
- extreme_value distribution
- f distribution
- factorial distribution
- gamma distribution
- gaussian distribution
- gaussian_tail distribution
- generalized_hypergeometric_b3 distribution
- generalized_waring distribution
- geometric distribution
- grassia distribution
- gumbel_1 distribution
- gumbel_2 distribution
- hyperbolic_secant distribution
- hypergeometric distribution
- inverse_gaussian distribution
- inverse_polya_eggenberger distribution
- lambda distribution
- laplace distribution

- levy distribution
- list distribution
- logarithmic distribution
- logistic distribution
- lognormal distribution
- mizutani distribution
- negative_binomial distribution
- negative_binomial_beta distribution
- normal distribution
- pareto distribution
- pascal distribution
- pearson distribution
- planck distribution
- poisson distribution
- polya distribution
- polya_aeppli distribution
- rayleigh distribution
- rayleigh_tail distribution
- singh_maddala distribution
- t distribution
- teichroew distribution
- triangular distribution
- trigamma distribution
- uniform distribution
- von_mises distribution
- wald distribution
- waring distribution
- weibull distribution
- yule distribution
- zipf distribution

2.3 Engines

Currently we have 3 engines implemented:

- linear_congruential
- mitchell_moore
- mt19937