Chapter 14

Random Number Generation and Statistical Distributions

In this chapter we are going to construct classes to help us encapsulate the generation of random numbers. Random number generators (RNG) are an essential tool in quantitative finance as they are necessary for Monte Carlo simulations that power numerical option pricing techniques. Other chapters have so far used RNGs in a procedural manner. In particular, we have utilised the Box-Muller technique to generate one or more random variables distributed as a standard Gaussian.

14.1 Overview

We will now show how to construct a random number generator class hierarchy. This allows us to separate the generation of random numbers from the Monte Carlo solvers that make use of them. It helps us reduce the amount of code we will need to write in the future, increases extensibility by allowing easy creation of additional random number generators and makes the code more maintainable.

There are further reasons to write our own random number generators:

- It allows us to make use of **pseudo-random numbers**. These are sequences of numbers that possess the correct statistical properties to "emulate" random numbers in order to improve the convergence rates of Monte Carlo simulations. The interface for random numbers and pseudo-random numbers is identical and we can hide away the details in the specific classes. In particular we can implement *low-discrepancy numbers* and *anti-thetic sampling* in this manner.

- Relying on the rand function provided with the C++ standard is **unreliable**. Not only is rand implementation specific, because it varies across multiple vendor compilers, but we are unaware of the efficiency of each implementation. This leads to difficulties in cross-platform testing as we cannot guarantee reproducibility.

- We are able to provide **multiple separate streams of random numbers** for different parts of our running program. The seed for the rand function, srand, is a global variable and hence will affect all components of our program, which is usually unwanted behaviour. By implementing our own RNG we avoid this issue.

14.2 Random Number Generator Class Hierarchy

Our random number generators will be formed from an *inheritance hierarchy*. We have already used this method when constructing PayOff classes for option pay-off functions. To form the hierarchy we will create an *abstract base class* that specifies the interface to the random number generator. All subsequent generators will inherit the interface from this class.

The primary considerations of this interface are as follows:

- Quantity or **dimension** of the generator: Many of the options pricers we have already created require more than a single random number in order to be accurately priced. This is the case for *path-dependent* options such as *Asians*, *Barriers* and *Lookbacks*. Thus our first consideration is to make sure that the generator provides a vector of random numbers, with the dimension specified at the creation of the instance.

- The supported **statistical distributions** from which to draw random variables: For options pricing, the two main statistical distributions of interest will be the *uniform distribution* and the *standard normal distribution* (i.e. the "Gaussian" distribution). Gaussian random draws are calculated from uniform random draws. We can use the statistical classes in order to obtain random draws from any particular distribution we wish, without modifying the RNG.

- We will need methods to support obtaining and setting the *random seed*, so that we can control which random numbers are generated and to ensure reproducibility across separate runs and platforms.

With those considerations in mind, let's create a simple abstract base class for our random number generator, in the file random.h:

```
#ifndef __RANDOM_H
#define __RANDOM_H

#include <vector>

class RandomNumberGenerator {
 protected:
  unsigned long init_seed;  // Initial random seed value
  unsigned long cur_seed;   // Current random seed value
  unsigned long num_draws;  // Dimensionality of the RNG

 public:
  RandomNumberGenerator(unsigned long _num_draws, unsigned long _init_seed)
    : num_draws(_num_draws), init_seed(_init_seed), cur_seed(_init_seed)
      {};
  virtual ~RandomNumberGenerator() {};

  virtual unsigned long get_random_seed() const { return cur_seed; }
  virtual void set_random_seed(unsigned long _seed) { cur_seed = _seed; }
  virtual void reset_random_seed() { cur_seed = init_seed; }
  virtual void set_num_draws(unsigned long _num_draws) { num_draws =
      _num_draws; }

  // Obtain a random integer (needed for creating random uniforms)
  virtual unsigned long get_random_integer() = 0;

  // Fills a vector with uniform random variables on the open interval
      (0,1)
  virtual void get_uniform_draws(std::vector<double>& draws) = 0;
};

#endif
```

Let's run through the code. Firstly, note that we have three **protected** member variables (which are all large **unsigned long** integers). cur_seed is the RNG current seed value. init_seed is the initial seed value, which does not change once the RNG has been instantiated. The current seed can only be reset to the initial seed. num_draws represents the *dimensionality* of the random number generator (i.e. how many random draws to create):

```
protected:
 unsigned long init_seed;  // Initial random seed value
 unsigned long cur_seed;   // Current random seed value
 unsigned long num_draws;  // Dimensionality of the RNG
```

Since we're creating an *abstract* base class is it a good idea to use **protected** data?

This is actually a contentious issue. Sometimes protected variables are frowned upon. Instead, it is argued that all data should be private and that accessor methods should be used. However, inherited classes -are- clients of the base class, just as "public" clients of the classes are. The alternative argument is that it is extremely convenient to use protected member data because it reduces the amount of cluttered accessor and modifier methods. For brevity protected member data has been used here.

Although the class will never be instantiated directly, it still has a constructor which must be called to populate the protected members. We use a member initialisation list to carry this out. We also create an empty method implementation for the constructor ({}), avoiding the need to create a random.cpp source file. Notice that we're setting the current seed to the initial seed as well.

```
RandomNumberGenerator(unsigned long _num_draws, unsigned long _init_seed)
    : num_draws(_num_draws), init_seed(_init_seed), cur_seed(_init_seed)
        {};
```

We then have four separate access and reset methods (all virtual), which get, set and reset the *random seed* and another which resets the number of random draws. They are all directly implemented in the header file, once again stopping us from needing to create a random.cpp source file:

```
virtual unsigned long get_random_seed() const { return cur_seed; }
virtual void set_random_seed(unsigned long _seed) { cur_seed = _seed; }
virtual void reset_random_seed() { cur_seed = init_seed; }
virtual void set_num_draws(unsigned long _num_draws) { num_draws =
    _num_draws; }
```

We now need a method to create a random integer. This is because subsequent random number generators will rely on transforming random **unsigned long**s into uniform variables on the open interval $(0, 1)$. The method is declared pure virtual as different RNGs will implement this differently. We don't want to "force" an approach on future clients of our code:

```
// Obtain a random integer (needed for creating random uniforms)
virtual unsigned long get_random_integer() = 0;
```

Finally we fill a supplied vector with uniform random draws. This vector will then be passed to a statistical distribution class in order to generate random draws from any chosen distribution that we implement. In this way we are completely separating the generation of the uniform random variables (on the open interval $(0, 1)$) and the draws from various statistical distributions. This maximises code re-use and aids testing:

```
// Fills a vector with uniform random variables on the open interval (0,1)
virtual void get_uniform_draws(std::vector<double>& draws) = 0;
```

Our next task is to implement a **linear congruential generator** algorithm as a means for creating our uniform random draws.

14.2.1 Linear Congruential Generators

Linear congruential generators (LCG) are a form of random number generator based on the following general recurrence relation:

$$x_{k+1} = g \cdot x_k modn$$

Where n is a prime number (or power of a prime number), g has *high multiplicative order modulo* n and x_0 (the initial seed) is co-prime to n. Essentially, if g is chosen correctly, all integers from 1 to $n - 1$ will eventually appear in a periodic fashion. This is why LCGs are termed *pseudo-random*. Although they possess "enough" randomness for our needs (as n can be large), they are far from truly random. We won't dwell on the details of the mathematics behind LCGs, as we will not be making strong use of them going forward in our studies. However, most system-supplied RNGs make use of LCGs, so it is worth being aware of the algorithm. The listing below (lin_con_gen.cpp) contains the implementation of the algorithm. If you want to learn more about how LCGs work, take a look at Numerical Recipes[20].

14.2.2 Implementing a Linear Congruential Generator

With the mathematical algorithm described, it is straightforward to create the header file listing (lin_con_gen.h) for the Linear Congruential Generator. The LCG simply inherits from the RNG abstract base class, adds a **private** member variable called max_multiplier (used for pre-computing a specific ratio required in the uniform draw implementation) and implements the two pure virtual methods that were part of the RNG abstract base class:

```
#ifndef __LINEAR_CONGRUENTIAL_GENERATOR_H
#define __LINEAR_CONGRUENTIAL_GENERATOR_H

#include "random.h"

class LinearCongruentialGenerator : public RandomNumberGenerator {
 private:
  double max_multiplier;

 public:
  LinearCongruentialGenerator(unsigned long _num_draws,
                              unsigned long _init_seed = 1);
  virtual ~LinearCongruentialGenerator() {};

  virtual unsigned long get_random_integer();
  virtual void get_uniform_draws(std::vector<double>& draws);
};

#endif
```

The source file (lin_con_gen.cpp) contains the implementation of the linear congruential generator algorithm. We make heavy use of *Numerical Recipes in C*[20], the famed numerical algorithms cookbook. The book itself is freely available online. We strongly suggest reading the chapter on random number generator (Chapter 7) as it describes many of the pitfalls with using a basic linear congruential generator, which is outside of the scope of this chapter. Here is the listing in full:

```
#ifndef __LINEAR_CONGRUENTIAL_GENERATOR_CPP
#define __LINEAR_CONGRUENTIAL_GENERATOR_CPP

#include "lin_con_gen.h"
```

```
// This uses the Park & Miller algorithm found in "Numerical Recipes in C"
// Define the constants for the Park & Miller algorithm

const unsigned long a = 16807;       // 7^5
const unsigned long m = 2147483647;  // 2^32 - 1 (and thus prime)

// Schrage's algorithm constants

const unsigned long q = 127773;
const unsigned long r = 2836;

// Parameter constructor
LinearCongruentialGenerator::LinearCongruentialGenerator(
    unsigned long _num_draws,
    unsigned long _init_seed
) : RandomNumberGenerator(_num_draws, _init_seed) {

  if (_init_seed == 0) {
    init_seed = 1;
    cur_seed = 1;
  }

  max_multiplier = 1.0 / (1.0 + (m-1));
}

// Obtains a random unsigned long integer
unsigned long LinearCongruentialGenerator::get_random_integer() {
  unsigned long k = 0;
  k = cur_seed / q;
  cur_seed = a * (cur_seed - k * q) - r * k;

  if (cur_seed < 0) {
    cur_seed += m;
  }

  return cur_seed;
}

// Create a vector of uniform draws between (0,1)
void LinearCongruentialGenerator::get_uniform_draws(std::vector<double>&
    draws) {
  for (unsigned long i=0; i<num_draws; i++) {
    draws[i] = get_random_integer() * max_multiplier;
  }
}

#endif
```

Firstly, we set all of the necessary constants (see [20] for the explanation of the chosen values). *Note that if we created another LCG we could inherit from the RNG base class and use different constants*:

```
// Define the constants for the Park & Miller algorithm

const unsigned long a = 16807;        // 7^5
const unsigned long m = 2147483647;   // 2^32 - 1

// Schrage's algorithm constants

const unsigned long q = 127773;
const unsigned long r = 2836;
```

Secondly we implement the constructor for the LCG. If the seed is set to zero by the client, we set it to unity, as the LCG algorithm does not work with a seed of zero. The max_mutliplier is a pre-computed scaling factor necessary for converting a random **unsigned long** into a uniform value on on the open interval $(0, 1) \subset \mathbb{R}$:

```
// Parameter constructor
LinearCongruentialGenerator::LinearCongruentialGenerator(
    unsigned long _num_draws,
    unsigned long _init_seed
) : RandomNumberGenerator(_num_draws, _init_seed) {

  if (_init_seed == 0) {
    init_seed = 1;
    cur_seed = 1;
  }

  max_multiplier = 1.0 / (1.0 + (m-1));
}
```

We now concretely implement the two pure virtual functions of the RNG base class, namely get_random_integer and get_uniform_draws. get_random_integer applies the LCG modulus algorithm and modifies the current seed (as described in the algorithm above):

```
// Obtains a random unsigned long integer
unsigned long LinearCongruentialGenerator::get_random_integer() {
  unsigned long k = 0;
  k = cur_seed / q;
  cur_seed = a * (cur_seed - k * q) - r * k;

  if (cur_seed < 0) {
    cur_seed += m;
  }

  return cur_seed;
}
```

get_uniform_draws takes in a vector of the correct length (num_draws) and loops over it converting random integers generated by the LCG into uniform random variables on the interval $(0, 1)$:

```
// Create a vector of uniform draws between (0,1)
void LinearCongruentialGenerator::get_uniform_draws(std::vector<double>&
    draws) {
  for (unsigned long i=0; i<num_draws; i++) {
    draws[i] = get_random_integer() * max_multiplier;
  }
}
```

This concludes the implementation of the linear congruential generator. The final component is to tie it all together with a main.cpp program.

14.2.3 Implementation of the Main Program

Because we have already carried out most of the hard work in random.h, lin_con_gen.h, lin_con_gen.cpp, the main implementation (main.cpp) is straightforward:

```
#include <iostream>
#include "lin_con_gen.h"

int main(int argc, char **argv) {
  // Set the initial seed and the dimensionality of the RNG
  unsigned long init_seed = 1;
  unsigned long num_draws = 20;
  std::vector<double> random_draws(num_draws, 0.0);

  // Create the LCG object and create the random uniform draws
  // on the open interval (0,1)
  LinearCongruentialGenerator lcg(num_draws, init_seed);
  lcg.get_uniform_draws(random_draws);

  // Output the random draws to the console/stdout
  for (unsigned long i=0; i<num_draws; i++) {
    std::cout << random_draws[i] << std::endl;
  }

  return 0;
}
```

Firstly, we set up the initial seed and the dimensionality of the random number generator. Then we pre-initialise the vector, which will ultimately contain the uniform draws. Then we instantiate the linear congruential generator and pass the random_draws vector into the get_uniform_draws method. Finally we output the uniform variables. The output of the code is as follows:

```
7.82637e-06
0.131538
0.755605
0.45865
0.532767
0.218959
```

```
0.0470446
0.678865
0.679296
0.934693
0.383502
0.519416
0.830965
0.0345721
0.0534616
0.5297
0.671149
0.00769819
0.383416
0.0668422
```

As can be seen, all of the values lie between $(0, 1)$. We are now in a position to utilise statistical distributions with the uniform random number generator to obtain random draws.

14.3 Statistical Distributions

One of the commonest concepts in quantitative finance is that of a *statistical distribution*. Random variables play a huge part in quantitative financial modelling. Derivatives pricing, cash-flow forecasting and quantitative trading all make use of statistical methods in some fashion. Hence, modelling statistical distributions is extremely important in C++.

Many of the chapters within this book have made use of random number generators in order to carry out pricing tasks. So far this has been carried out in a *procedural* manner. Functions have been called to provide random numbers without any data encapsulation of those random number generators. The goal of this chapter is to show you that it is beneficial to create a *class hierarchy* both for statistical distributions and random number generators, separating them out in order to gain the most leverage from code reuse.

In a nutshell, we are splitting the generation of (uniform integer) random numbers from draws of specific statistical distributions, such that we can use the statistics classes elsewhere without bringing along the "heavy" random number generation functions. Equally useful is the fact that we will be able "swap out" different random number generators for our statistics classes for reasons of reliability, extensibility and efficiency.

14.3.1 Statistical Distribution Inheritance Hierarchy

The inheritance hierarchy for modelling of statistical distributions is relatively simple. Each distribution of interest will share the same interface, so we will create an *abstract base class*, as was carried out for the PayOff hierarchy. We are primarily interested in modelling continuous probability distributions for the time being.

Each (continuous) statistical distribution contains the following properties to be modelled:

- **Domain Interval** - The interval subset of $\mathbb{R}$ with which the distribution is defined for
- **Probability Density Function** - Describes the frequency for any particular value in our domain
- **Cumulative Density Function** - The function describing the probability that a value is less than or equal to a particular value
- **Expectation** - The expected value (the mean) of the distribution
- **Variance** - Characterisation of the spread of values around the expected value
- **Standard Deviation** - The square root of the variance, used because it possesses the same units as the expected value, unlike the variance

We also wish to produce a sequence of random draws from this distribution, assuming a sequence of random numbers is available to provide the "randomness". We can achieve this in two ways. We can either use the *inverse cumulative distribution function* (also known as the *quantile function*), which is a property of the distribution itself, or we can use a custom method (such as Box-Muller). Some of the distributions do not possess an analytical inverse to the CDF and hence they will need to be approximated numerically, via an appropriate algorithm. This calculation will be encapsulated into the class of the relevant inherited distribution.

Here is the partial header file for the StatisticalDistribution abstract base class (we will add extra distributions later):

```
#ifndef __STATISTICS_H
#define __STATISTICS_H

#include <cmath>
#include <vector>

class StatisticalDistribution {
```

```
public:
  StatisticalDistribution();
  virtual ~StatisticalDistribution();

  // Distribution functions
  virtual double pdf(const double& x) const = 0;
  virtual double cdf(const double& x) const = 0;

  // Inverse cumulative distribution functions (aka the quantile function)
  virtual double inv_cdf(const double& quantile) const = 0;

  // Descriptive stats
  virtual double mean() const = 0;
  virtual double var() const = 0;
  virtual double stdev() const = 0;

  // Obtain a sequence of random draws from this distribution
  virtual void random_draws(const std::vector<double>& uniform_draws,
                            std::vector<double>& dist_draws) = 0;
};

#endif
```

We've specified pure virtual methods for the probability density function (pdf), cumulative density function (cdf), inverse cdf (inv_cdf), as well as descriptive statistics functions such as mean, var (variance) and stdev (standard deviation). Finally we have a method that takes in a vector of uniform random variables on the open interval $(0, 1)$, then fills a vector of identical length with draws from the distribution.

Since all of the methods are pure virtual, we only need a very simple implementation of a source file for this class, since we are simply specifying an interface. However, we would like to see a concrete implementation of a particular class. We will consider, arguably, the most important distribution in quantitative finance, namely the standard normal distribution.

14.3.2 Standard Normal Distribution Implementation

Firstly we'll briefly review the formulae for the various methods we need to implement for the standard normal distribution. The probability density function of the standard normal distribution is given by:

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

The cumulative density function is given by:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt$$

The inverse cumulative density function of the standard normal distribution (also known as the *probit function*) is somewhat more involved. No analytical formula exists for this particular function and so it must be approximated by numerical methods. We will utilise the Beasley-Springer-Moro algorithm, found in Korn[15].

Given that we are dealing with the standard normal distribution, the mean is simply $\mu = 0$, variance $\sigma^2 = 1$ and standard deviation, $\sigma = 1$. The implementation for the header file (which is a continuation of statistics.h above) is as follows:

```
class StandardNormalDistribution : public StatisticalDistribution {
 public:
  StandardNormalDistribution();
  virtual ~StandardNormalDistribution();

  // Distribution functions
  virtual double pdf(const double& x) const;
  virtual double cdf(const double& x) const;

  // Inverse cumulative distribution function (aka the probit function)
  virtual double inv_cdf(const double& quantile) const;

  // Descriptive stats
  virtual double mean() const;   // equal to 0
  virtual double var() const;    // equal to 1
  virtual double stdev() const;  // equal to 1

  // Obtain a sequence of random draws from the standard normal
      distribution
  virtual void random_draws(const std::vector<double>& uniform_draws,
                            std::vector<double>& dist_draws);
};
```

The source file is given below:

```
#ifndef __STATISTICS_CPP
#define __STATISTICS_CPP

#define _USE_MATH_DEFINES

#include "statistics.h"
#include <iostream>

StatisticalDistribution::StatisticalDistribution() {}
StatisticalDistribution::~StatisticalDistribution() {}

// Constructor/destructor
StandardNormalDistribution::StandardNormalDistribution() {}
StandardNormalDistribution::~StandardNormalDistribution() {}

// Probability density function
double StandardNormalDistribution::pdf(const double& x) const {
  return (1.0/sqrt(2.0 * M_PI)) * exp(-0.5*x*x);
}

// Cumulative density function
double StandardNormalDistribution::cdf(const double& x) const {
  double k = 1.0/(1.0 + 0.2316419*x);
  double k_sum = k*(0.319381530 + k*(-0.356563782 + k*(1.781477937 +
      k*(-1.821255978 + 1.330274429*k))));

  if (x >= 0.0) {
    return (1.0 - (1.0/(pow(2*M_PI,0.5)))*exp(-0.5*x*x) * k_sum);
  } else {
    return 1.0 - cdf(-x);
  }
}
```

```
// Inverse cumulative distribution function (aka the probit function)
double StandardNormalDistribution::inv_cdf(const double& quantile) const {
  // This is the Beasley-Springer-Moro algorithm which can
  // be found in Glasserman [2004]. We won't go into the
  // details here, so have a look at the reference for more info
  static double a[4] = {   2.50662823884,
                         -18.61500062529,
                          41.39119773534,
                         -25.44106049637};

  static double b[4] = {  -8.47351093090,
                          23.08336743743,
                         -21.06224101826,
                           3.13082909833};

  static double c[9] = {0.3374754822726147,
                        0.9761690190917186,
                        0.1607979714918209,
                        0.0276438810333863,
                        0.0038405729373609,
                        0.0003951896511919,
                        0.0000321767881768,
                        0.0000002888167364,
                        0.0000003960315187};

  if (quantile >= 0.5 && quantile <= 0.92) {
    double num = 0.0;
    double denom = 1.0;

    for (int i=0; i<4; i++) {
      num += a[i] * pow((quantile - 0.5), 2*i + 1);
      denom += b[i] * pow((quantile - 0.5), 2*i);
    }
    return num/denom;

  } else if (quantile > 0.92 && quantile < 1) {
    double num = 0.0;

    for (int i=0; i<9; i++) {
      num += c[i] * pow((log(-log(1-quantile))), i);
    }
    return num;

  } else {
    return -1.0*inv_cdf(1-quantile);
  }
}

// Expectation/mean
double StandardNormalDistribution::mean() const { return 0.0; }

// Variance
double StandardNormalDistribution::var() const { return 1.0; }

// Standard Deviation
double StandardNormalDistribution::stdev() const { return 1.0; }

// Obtain a sequence of random draws from this distribution
void StandardNormalDistribution::random_draws(
    const std::vector<double>& uniform_draws,
    std::vector<double>& dist_draws
) {
  // The simplest method to calculate this is with the Box-Muller method,
  // which has been used procedurally in many other chapters

  // Check that the uniform draws and dist_draws are the same size and
  // have an even number of elements (necessary for B-M)
  if (uniform_draws.size() != dist_draws.size()) {
    std::cout << "Draw vectors are of unequal size." << std::endl;
    return;
  }

  // Check that uniform draws have an even number of elements (necessary
      for B-M)
  if (uniform_draws.size() % 2 != 0) {
    std::cout << "Uniform draw vector size not an even number." << std::
        endl;
    return;
  }

  // Slow, but easy to implement
  for (int i=0; i<uniform_draws.size() / 2; i++) {
    dist_draws[2*i] = sqrt(-2.0*log(uniform_draws[2*i])) *
        sin(2*M_PI*uniform_draws[2*i+1]);
    dist_draws[2*i+1] = sqrt(-2.0*log(uniform_draws[2*i])) *
        cos(2*M_PI*uniform_draws[2*i+1]);
  }

  return;
}
#endif
```

Some of the implementations are discussed briefly here. The cumulative distribution function (cdf) is referenced from Joshi[12]. It is an approximation, rather than closed-form solution. The inverse CDF (inv_cdf) makes use of the Beasley-Springer-Moro algorithm, which was implemented via the algorithm given in Korn[15]. A similar method can be found in Joshi[11]. Once again the algorithm is an approximation to the real function, rather than a closed form solution. The final method is random_draws. In this instance we are using the Box-Muller algorithm. However, we could instead utilise the more efficient Ziggurat algorithm, although we won't do so here.

14.3.3 The Main Listing

We will now utilise the new statistical distribution classes with a simple random number generator in order to output statistical values:

```
#include "statistics.h"
#include <iostream>
#include <vector>

int main(int argc, char **argv) {
  // Create the Standard Normal Distribution and random draw vectors
  StandardNormalDistribution snd;
  std::vector<double> uniform_draws(20, 0.0);
  std::vector<double> normal_draws(20, 0.0);

  // Simple random number generation method based on RAND
  for (int i=0; i<uniform_draws.size(); i++) {
    uniform_draws[i] = rand() / static_cast<double>(RAND_MAX);
  }

  // Create standard normal random draws
  // Notice that the uniform draws are unaffected. We have separated
  // out the uniform creation from the normal draw creation, which
  // will allow us to create sophisticated random number generators
  // without interfering with the statistical classes
  snd.random_draws(uniform_draws, normal_draws);

  // Output the values of the standard normal random draws
  for (int i=0; i<normal_draws.size(); i++) {
    std::cout << normal_draws[i] << std::endl;
  }

  return 0;
}
```

The output from the program is as follows (a sequence of normally distributed random variables):

```
3.56692
3.28529
0.192324
-0.723522
1.10093
0.217484
-2.22963
-1.06868
-0.35082
0.806425
```

```
-0.168485
-1.3742
0.131154
0.59425
-0.449029
-2.37823
0.0431789
0.891999
0.564585
1.26432
```

Now that we have set up the inheritance hierarchy, we could construct additional (continuous) statistical distributions, such as the *log-normal distribution*, the *gamma distribution* and the *chi-square distribution*.