Random Variate Generator

Joseph R. Laracy
Seton Hall University
400 South Orange Avenue
South Orange, NJ 07079
laracyjo@shu.edu

Abstract

As the use of computer simulation grows in a variety of science and engineering fields, the quality of random variate generators becomes increasingly important. Unfortunately, a number of standard implementations are grossly inadequate and exhibit poor statistical properties. This paper presents a software pattern for efficiently implementing an extensible, high quality random variate generator.

Keywords: Modeling and Simulation, Patterns, Object-Oriented Design Methods

Random Variate Generator

Intent

Software patterns are a great way to share well trusted solutions to common development problems [1]. This pattern solves the problem that many scientists and engineers encounter when they use inadequate built in/library random number generators that do not exhibit the required mathematical properties. Instead of reflexively calling the included random number generation routine, developers who are familiar with this pattern can often quickly code a much better generator. The two objectives of this pattern are:

1. To provide a random number generator that efficiently produces samples that are
    a. Uniformly distributed
    b. Independent from one another
2. Through the use of inheritance, to enable developers to change the random number generating algorithm *and* random variate distribution.

NOTE: "A random variate is a variable generated from uniformly distributed pseudorandom numbers. Depending on how they are generated, a random variate can be uniformly or non-uniformly distributed. Random variates are frequently used as the input to simulation models" [2].

Motivation

One of the most powerful uses of high performance computing is in the field of modeling and simulation. Computers are being used to gain a deeper understanding and even predictive power in problems ranging from traffic congestion to weather to nuclear physics. Unfortunately, many scientists and engineers use substandard random number generators to create random variates for their simulations.

As early at 1981, Knuth identified that the IBM SYSTEM/360 injected an algorithm into industry that places an overemphasis on computational efficiency and reliance on the SYSTEM/360 architecture, and no emphasis on statistical properties. This code relies on an expected overflow in the 360 arithmetic unit. For decades, programmers have blindly used this algorithm not knowing the underlying assumptions behind it [3].

This problem became apparent recently to the financial modeling community with the first release of Microsoft Excel 2003 [4]. The RAND() function provided in Excel was supposed to generate a number between 0 and 1. However, it produced a distribution which included negative numbers as well. Additionally, those negative samples were not distributed randomly by any definition. This bug had a huge impact on financial institution that ran an Excel shop. The figure below was produced by generating 100,000 samples [5].

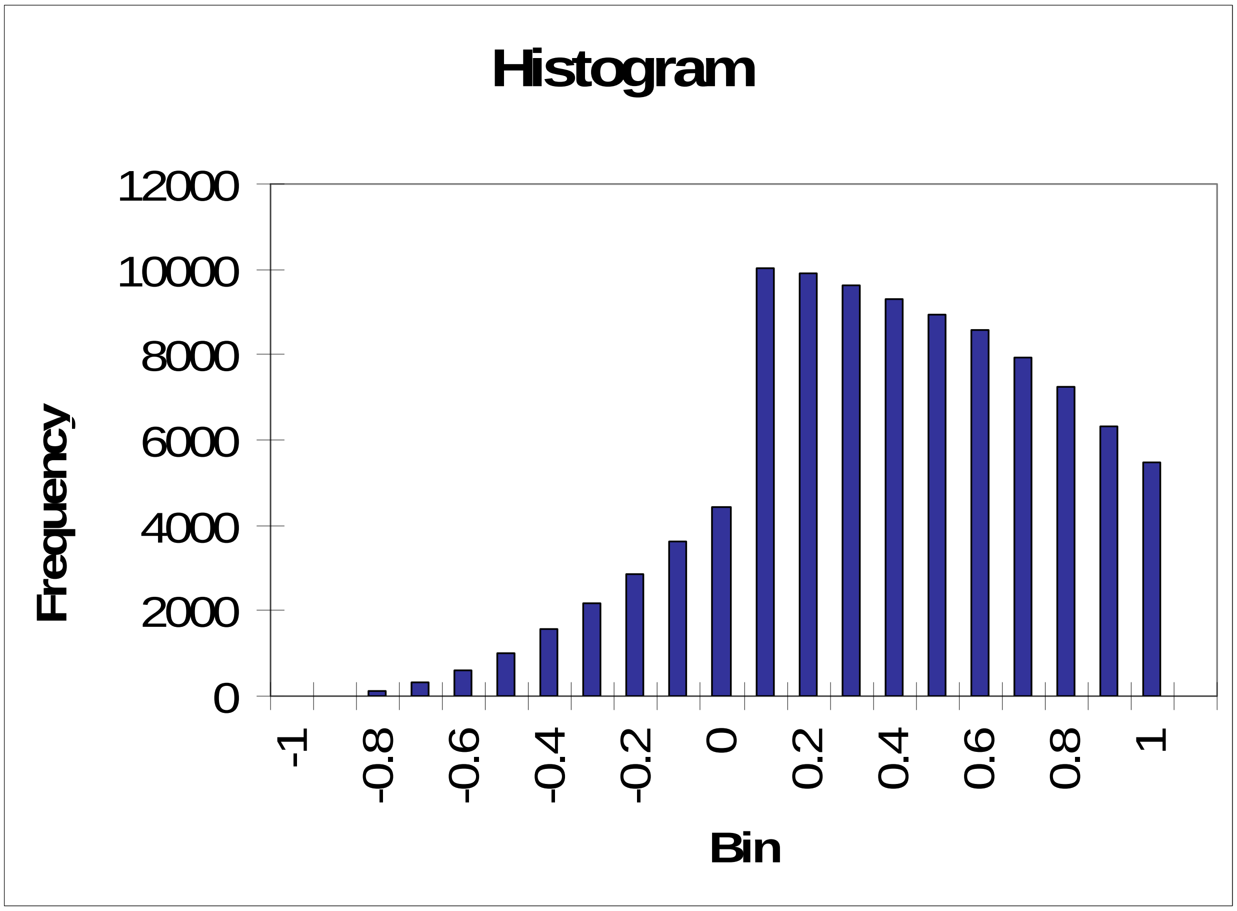


Figure 1. Histogram from Excel 2003 RAND().

Of course, there are a variety of different techniques for random number generation and certain specialists must consider the trade-offs among computational efficiency and statistical properties of the generator under various use cases. However, Park and Miller [6] point out that for most scientists and engineers:

> convenient and frequent access to a verifiably good random number generator is fundamentally important. If you have need for a random number generator, particularly one that will port to a wide variety of systems, and if you are not a specialist in random number generation and do not want to become one, use the minimal standard [i.e. algorithm provided in this article]. It should not be presumed that it is easy to write a better one.

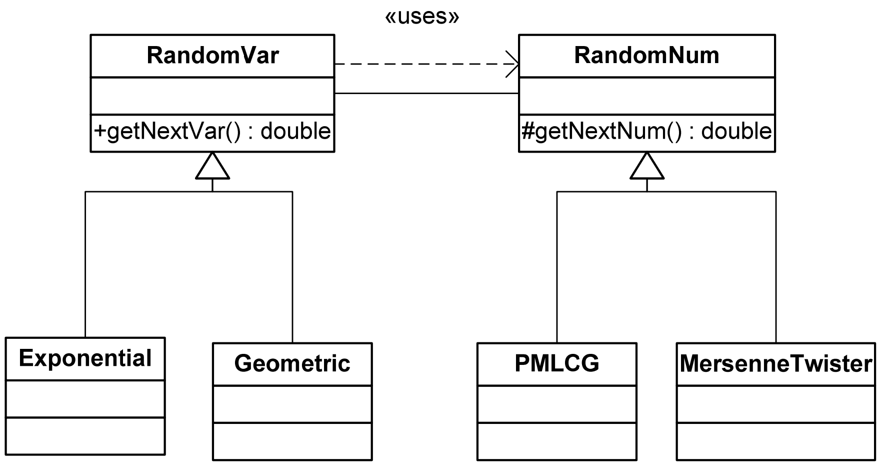
This pattern enables the typical user to implement the "minimal standard" generator algorithm and then develop particular distributions, e.g. Poisson, as well as supports the specialist who wants to switch generators depending on use case.

Applicability

Because of the poor quality of random number generators in the past, no algorithms or implementation should be blindly trusted. In just a few lines of code, any developer can implement a prime modulus linear congruential generator with much better statistical properties than other pre-packaged solutions. The author makes no claim about the applicability of this solution to cryptographic applications. Special techniques are employed for random numbers used in cryptography. The focus of this solution is simulation modeling:

- Continuous stochastic simulation (e.g. heat flow problems)
- Discrete event stochastic simulation (e.g. queuing problems)

Structure



The getNextVar() function is called in the program to get the next random variate in the distribution.

Participants

RandomVar class is the abstract super class of the random variate generator. Subclasses implement the particular distributions that may be desired, such as geometric or exponential.

The Exponential and Geometric classes are example subclasses of RandomVar.

RandomNum class is the abstract super class of the random number generator. Subclasses implement the particular algorithm, such as a prime modulus linear congruential (PMLGC) or Mersenne Twister.

The PMLGC and MersenneTwister classes are example subclasses of RandomNum.

RandomVar

- The getNextVar function is called on an instance of this class to get the next random variate in the sequence.

Exponential / Geometric

- An instance of this class takes the uniform distribution generated by RandomNum and performs the transformation to create a random variate.

RandomNum

- A RandomVar object creates an instance of this class and calls the getNextNum function to get the next random number from the uniform distribution.

PMLGC / MersenneTwister

- An instance of this class creates a uniform distribution of random numbers.

Collaborations

- RandomNum objects provide random numbers to RandomVar objects which generate the desired distribution.

Consequences

1. By separating the random number and random variate generation modules, new random number generators can be experimented with and tested.
2. By implementing particular random number generation algorithms in subclasses, the state-of-the-art algorithms can be added to an existing system without changing any other modules.
3. By implementing the random variate generation in subclasses, developers can very easily change distributions by creating another object for the desired distribution.

Implementation

1. The Park and Miller algorithm is provided below for a PMLGC. It has a period of m-1 and empirical studies indicate good statistical properties. Most simulations can use this algorithm to implement random numbers. A C++ implementation of the algorithm is provided in the next section. For information on the theory behind this algorithm , see [6].

2. If a greater period length is necessary, e.g. 2^{56}, Combined PMLCG can be employed. This extension is portable and simple to implement, but not fast. See page 186 in Volume 2 of [3] for more information. However, for simulations that require a “super astronomical” number of random numbers, the Mersenne Twister may be used. It has a period of $2^{19937} - 1$ and 623-dimensional independence. With 32-bit accuracy, only 624 words are necessary. For more information on the theory and implementation of this algorithm, see [7].
3. Geometric random variates are generated by taking the U ~ U(0,1) and returning *floor(ln U / ln (1-p))*.
4. Exponential random variates are generated by taking the U ~ U(0,1) and returning $-\beta ln U$ where β is the mean.
5. For details on implementing other probability distribution functions, see [8].

Sample Code and Usage

```
//Park and Miller Algorithm (1988) for PMLCG

 int m = 2147483647; // 2^31 – 1
 int a = 48271;
 int q = floor(m/a);
 int r = m % a;
 int Z = 456;

 for (int i = 0; i <numSamples; i++)
 {
          g[i] = a*(Z[i] % q) – r * floor(Z[i]/q);
          if g[i] >= 0
                   Z[i+1] = g[i];
          else
                   Z[i+1] = g[i] + m;
          Z[i+1] = Z[i+1]/m;                    //scale to 0 → 1
 }
//END
```

Known Uses

Some of the ideas contained in this pattern are used in object oriented simulation packages as well as custom simulation projects. Its structured programming analog is even more common. This pattern shares the spirit of the Gang of Four Strategy pattern. The intent of the Strategy pattern is to “define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” [9]

References

[1] M. Kircher, M. Völter, "Software Patterns," *IEEE Software*, vol. 24, pp. 28-30, 2007.

[2] F. Neelamkavil, *Computer Simulation and Modeling*. New York: Wiley, 1987.

[3] D. E. Knuth, *The Art of Computer Programming*. Reading, MA: Addison Wesley, 1981.

[4] K. B. Keeling, Robert J. Pavur, "Numerical Accuracy Issues in Using Excel for Simulation Studies," in *Proceedings of the 2004 Winter Simulation Conference*, Washington, DC, 2004.

[5] R. Engelbrecht-Wiggans, "Histogram from Excel 2003 RAND()," Champaign, IL, 2004.

[6] S. Park, K. Miller, "Random Number Generators: Good Ones Are Hard To Find," *Communications of the ACM*, vol. 31, 1988.

[7] M. Matsumoto, T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, 1998.

[8] W. Kelton, A. Law, *Simulation Modeling and Analysis*. New York: McGraw Hill, 2000.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*. Reading, MA: Addison Wesley, 1997.