OBJECT ORIENTED RANDOM NUMBER GENERATORS

Joseph A Fisher

Institute for Simulation and Training
University of Central Florida
Orlando, Florida 32826

ABSTRACT

An object oriented variate generator class was developed using the Prime Modulus Multiplicative Congruential Generator (PMMCG). The class provides the 18 most popular distributions and should prove useful to the simulationist.

INTRODUCTION

The ability to generate good random numbers is a principal constraint in the success of any stochastic simulation. Scientists and engineers have long used random numbers in their analyses but they have not always checked the accuracy of their generators. Donald Knuth [1981, p. 176], a Herculean contributor to scientific computing once said, "... look at the subroutine library of each computer installation in your organization, and replace the random number generators by good ones. Try to avoid being too shocked at what you find."

Object-Oriented Programming (OOP) has recently attracted a great deal of attention in the programming community. Encapsulation and inheritance properties of OOP provide greatly improved code reuse and program efficiency. OOP is based on the creation of objects. The object is an entity which can be abstracted from some real world concept. Object oriented programming and simulation are very similar in that they both model real world objects. Thus, creating objects that generate random numbers fits in well with object oriented simulation.

OOP advances on procedural based programming languages by allowing operations on defined data types. Each data type becomes a unique instance when it is instantiated. Instantiating a random number data type creates a random number object that is unique and not dependent on other similar data types. Thus, unlike many procedural based programming languages that have random number generators with stream dependencies, OOP random number objects have no stream dependencies. One could say that random number objects pull from their own stream whereas procedural based generators pull from the same stream.

This paper first discusses the basic concepts of random number generators, then an implementation of an object oriented random number generator class is presented. It is not the intent of this paper to provide a detailed discussion of random number generators; the literature provides much discussion on efficiency, randomness, and theoretical properties.

Proceedings Of The 15th Annual Conference on Computers and Industrial Engineering
March 8 - 10, 1993., Cocoa Beach, Florida., USA

OVERVIEW OF RANDOM NUMBER GENERATORS

Prime Modulus generators were first suggested by Lehmer [1951]. He presented an equation of the form

$$f(z) = (a * z) \bmod m \qquad\qquad U(0,1)_n = z_n / m$$

multiplier: a -- is an integer in the range [2, 3, ..., m - 1]
seed: z -- is an integer in the range [1, 2, ..., m - 1]
modulus: m -- is a large prime integer
uniform zero-one: U(0,1) -- is a real number between zero and one

where subsequent integers are generated by $z_{n+1} = f(z_n)$. If proper values of multiplier and modulus are chosen, then a full period generator will result. Since long periods are desirable, and are bounded by the modulus, a large prime like $2^{31} - 1$ is chosen. Lehmer suggested this modulus in 1951 and today it fits very well with 32-bit arithmetic. Selection of the multiplier is a more complicated matter if a full period is desired. Fortunately, very good full period multipliers have been identified by Fishman and Moore [1985] for generators with modulus $2^{31} - 1$.

Fishman and Moore conducted an exhaustive search of all the possible multipliers for the PMMCG with modulus $2^{31} - 1$. First, they identified all the primitive roots of m since these are the only values which will result in a full period generator. There are 534.6 million multipliers, or approximately 25% of all possible multipliers which will produce a full period generator. Second, they tested these multipliers for randomness. After their exhaustive search, they concluded that only 410 passed a battery of statistical tests which would classify them as acceptable. Fishman and Moore published 205 of the 410 multipliers in their paper. The other 205 can be obtained from the published 205 and an algorithm. Fishman and Moore recommend 5 of the 205 as having outstanding statistical properties (Figure 1).

Fishman and Moore's Recommended Multipliers

for PMMCG

multipliers	seeds [1, 2, ..., 2147483646]
950706376	
742938285	seed(i+1) = (multiplier * seed(i)) mod (2147483647)
1226874159	
62089911	U(0,1) = seed(i+1) / (2147483647)
1343714438	

Figure 1

Selecting a seed is somewhat arbitrary as long as it is within the proper range of values. Fishman [1978] provides tables of random number seeds. He suggests that seeds should be separated by 100,000. Kleijnen [1986] discusses the selection of random number seeds in practice.

THE RANDOM NUMBER GENERATOR CLASS

The random number generator classes created consist of one parent class and 18 children classes. The parent class Rand generates the U(0, 1) random number and each child class transforms this number into the desired distribution. Rand is an abstract class and it implements a PMMCG.

Rand and its children classes were designed to be flexible, easily understandable, small, and efficient. Maximum efficiency can be achieved through the use of inline functions. Each instantiation of a random number object is unique and the only relationship with other random number objects is in the use of the static default seed. Rand provides high level data and functionality to each of the children classes. Figure 2 displays the parent class Rand and its children classes. Each of the children classes can be instantiated to create a unique random number object.

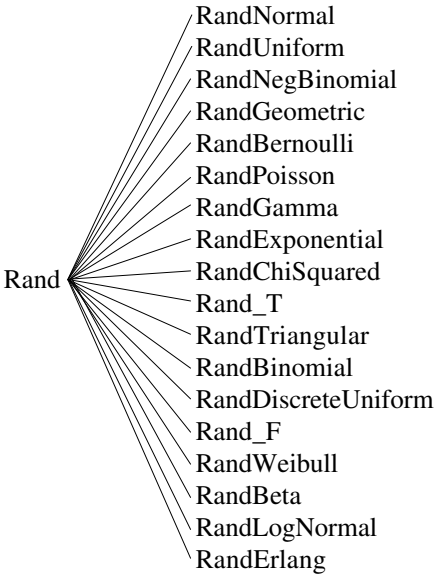

Figure 2

To create a random number object requires the dynamic or static instantiation of one of the children objects. To create a exponentially distributed random number object with a mean of 5.5 and default seed, a programmer would write the following code.

```
RandExponential RNG( 5.5 );
```

Generating random numbers requires a simple message to RNG as follows.

```
RNG.RandValue();
```

Their are two methods that are called when RNG is sent the message RandValue. The first method RandValue will request the next U(0,1) from the parent class Rand and then transform this into an exponentially distributed random variable. Figures 3 and 4 present the C++ code.

```
double Rand::GetUnif01( void )
  {
  seed = (long) fmodl( ( (long double) multiplier * (long double) seed ), (long double) LONG_MAX );

  return( (double) seed / (double) LONG_MAX );
  }
```

Figure 3

```
double RandExponential::RandValue( void )
  {
  return ( - mean * log( GetUnif01() ) );
  }
```

Figure 4

CONCLUSIONS

This paper outlined a well tested, industry accepted random number generator algorithm. Its implementation in the high level programming language C++ produced 18 useful generator classes. These classes will be useful to programmers and simulationists who want to have control over their random number generator. Source code and example programs are available via anonymous ftp at bikini.cis.ufl.edu (128.227.224.1).

REFERENCES

Fishman, G. S., "Principals of Discrete Event Simulation", John Wiley & Sons, Inc., New York, 1978.

Fishman, G. S., and Moore, III, L. S. "An exhaustive analysis of multiplicative congruential random number generators with modulus 2^{31} - 1", SIAM J. Sci. Comput, Vol. 7, No. 1, January 1986, pgs., 24 - 45.

Kleijnen, J. P. C., "Selecting random number seeds in practice", Simulation, Vol. 47, No. 1, July 1986, pgs., 15 - 17.

Knuth, D. E., "The art of computer programming Vol. 2: Seminumerical Algorithms", 2nd ed., Addison-Wesley, Reading, MA., 1981.

Lehmer, D. H., "Mathematical methods in large-scale computing units", Annu. Comput. Lab. Harvard Univ. 1951, pgs., 141 - 146.