FE545 Design, Patterns and Derivatives Pricing

A Random Numbers Class

Steve Yang

Stevens Institute of Technology

steve.yang@stevens.edu

02/27/2024

Overview

Strategy vs. Decorator Patter

- The **strategy pattern** allows you to change the implementation of something used at runtime.
  1. Defines a family of algorithms,
  2. Encapsulates each algorithm, and
  3. Makes the algorithms interchangeable within that family.
- The **decorator pattern** allows you augment (or add to) existing functionality with additional functionality at run time.
  1. Add additional functionalities/responsibilities dynamically
  2. Remove functionalities/responsibilities dynamically
  3. Avoid too much of sub-classing to add additional responsibilities.
- Strategy lets you change the guts of an object. Decorator lets you change the skin.

Random Number Generation

In a typical stochastic simulation, randomness is introduced into simulation models via independent uniformly distributed random variables. These random variables are then used as building blocks to simulate more general stochastic systems.

For example, consider a circle inscribed in a unit square. Given that the circle and the square have a ratio of areas that is $\pi/4$, the value of π can be approximated using a Monte Carlo method.

i. In the early days of simulation, randomness was generated by *manual* techniques, such as coin flipping, dice rolling, card shuffling, and roulette spinning.
ii. Later on, *physical devices*, such as noise diodes and Geiger counters, were attached to computers for the same purpose.
iii. The prevailing belief held that only mechanical or electronic devices could produce truly random sequences..

Monte Carlo Simulation Example

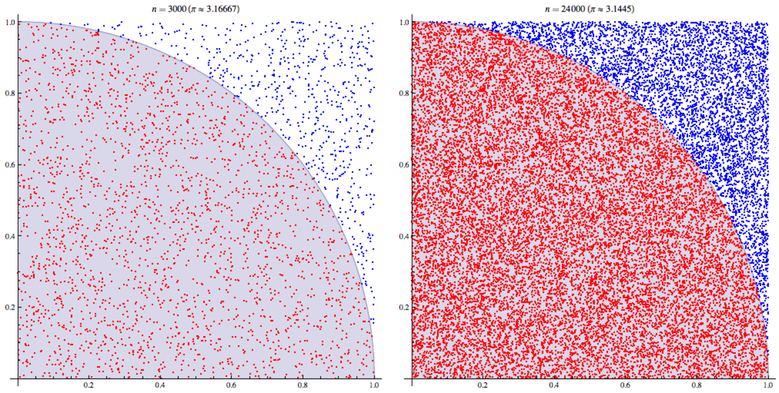


Figure: Monte Carlo Simulation to Calculating π.

Pseudorandom

- Although mechanical or electronic device based random number generators are still used in gambling and lotteries, these were generally abandoned by the computer-simulation community for several reasons:
  (a) Mechanical methods were too slow for general use;
  (b) The generated sequences cannot be reproduced;
  (c) It has been found that the generated numbers exhibit both bias and dependence.
- Most of today's random number generators are not based on physical devices, but on simple algorithms that can be easily implemented on a computer.
- They are fast, require little storage space, and can readily reproduce a given sequence of random numbers.
- More importantly, a good random generator captures all the important statistical properties of true random sequences.

Random Number Class

- We cannot expect any consistency across compilers. If we decide to test our code by running it on multiple platforms, we can expect to obtain differing streams of random numbers and while our Monte Carlo simulations should still converge to the same number.
- We do not know how good the compiler's implementation is. Either we have to get hold of technical documentation for every compiler we use and make sure that the implementation have done a good job, or we have to run a number of statistical tests to ensure that *rand* is up to the job.
- For most simulations, we will actually need many random draws for each path, and so it is not enough for us to check that single draws do a good job of simulating the uniform distribution; instead we need a large number of successive draws to do a good job of filling out the unit hypercube, which is much harder.

Random Number Class

- Another advantage of using a class is that we can decorate the class in whatever the way we want. For example, suppose we wish to use antithetic sampling. We could write a decorator class antithetic sampling.
- A further reason to use a class is that we might decide not to use psudo-random (i.e. congruential random number generator) but low-discrepancy numbers instead.
- We might replace the psudo-random number with quantum random number generator (QRNG).Their inner working can be clearly modelized and controlled to always produce high quality randomness.
- For financial applications, we will want standard Gaussian draws more often than uniforms so we will want a method of obtaining them instead. In fact, we can separate out the creation of the uniforms and their conversion into Gaussians.

Design Considerations

- We want the possibility of having many random number generators and we want to be able to add new ones without recoding, we use an abstract base class to specify an interface.
- We need to define dimensionality which is the number of random draws needed to simulate one path. This number is equal to the number of variables of integration in the underlying integral which we are trying to approximate.
- It is generally a good practice to obtain all the draws necessary for a path in one go. This has the added advantage that a random number generator can protest (i.e. throw an error) if it is being used beyond its dimensional specification.
- Occasionally, we wish to be sure of having a different stream of random numbers. For example, we generally use one set of random numbers to optimize parameters for the strategy, and then having chosen the strategy we run a second simulation with different random numbers to estimate the price.

The Base Class

Random2.h

```
#include "Arrays.h" // use MJArray

class RandomBase
{
public:
    RandomBase(unsigned long Dimensionality_);

    inline unsigned long GetDimensionality() const; // simple method, no need t
    virtual RandomBase* clone() const=0;

    // pure virtual methods
    virtual void GetUniforms(MJArray& variates)=0;// reference to a array, do n
    virtual void Skip(unsigned long numberOfPaths) = 0;
    virtual void SetSeed(unsigned long Seed) = 0;
    virtual void Reset() = 0;

    virtual void GetGaussians(MJArray& variates); // make virtual to allow anot
    virtual void ResetDimensionality(unsigned long NewDimensionality);

private:
    unsigned long Dimensionality;
};
```

Linear Congruential Generators

- The most common methods for generating pseudorandom sequences use the so-called *linear congruential generators*. They generate a deterministic sequence of numbers by means of the recursive formula:

$$\mathbf{X}_{i+1} = a\mathbf{X}_i + c \text{ (mod m)}. \quad (1)$$

where the initial value, $\mathbf{X}_0$ is called the *seed* and the a, c, and m (all positive integers) are called the *multiplier*, the *increment* and *modulus*, respectively. Not that applying the modulo-m operator means that $a\mathbf{X}_i + c$ is divided by m, and the reminder is taken as the value for $\mathbf{X}_{i+1}$. Thus, each $\mathbf{X}_i$ can only assume a value from the set $0, 1, 2, ..., m-1$, and the quantities

$$U_i = \frac{\mathbf{X}_i}{m} \quad (2)$$

called *pseudorandom numbers*, constitute approximations to a true sequence of uniform random variables.

Multiplicative Congruential Generators

- Not that the sequence $\mathbf{X}_0, \mathbf{X}_1, \mathbf{X}_2, \ldots$ will repeat itself after at most m steps and will therefore be periodic, with period not exceeding m.
- In the special case where $c = 0$, formula 1 simply reduces to

$$\mathbf{X}_i = a\mathbf{X}_i \text{ (mod m)}. \tag{3}$$

Such a generator is called a *multiplicative congruential generator*.

- It is readily seen that *an arbitrary choice of* $\mathbf{X}_0, a, c,$ *and* m *will not lead to a pseudorandom sequence with good statistical properties*. In fact, number theory has been used to show that only a few combinations of these produce satisfactory results.

Multiplicative Congruential Generators

- In computer implementations, m is selected as a large prime number that can be accommodated by the computer word size. For example, in a binary 32-bit word computer, statistically acceptable generators can be obtained by choosing $m = 2^{31} - 1$ and $a = 7^5$, provided that the first bit is a sign bit. A 64-bit or 128-bit word computer will naturally yield better statistical results.
- Formulas 1, 2, and 3 can be readily extended to pseudorandom vector generation.

$$\mathbf{X}_i = \mathbf{A}\mathbf{X}_i \ (\mathrm{mod}\mathbf{M}). \tag{4}$$

and

$$\mathbf{U}_i = \mathbf{M}^{-1}\mathbf{X}_i. \tag{5}$$

where **A** is a nonsingular *nxn* matrix, **M** and **X** are *n*-dimensional vectors.

Multiplicative Congruential Generators

- The Park–Miller random number generator (after Stephen K. Park and Keith W. Miller), is a type of linear congruential generator (LCG) that operates in multiplicative group of integers modulo n.
- The general formula is:

$$X_{k+1} = a \cdot X_k \mod m. \tag{6}$$

  where the modulus m is a prime number or a power of a prime number, the multiplier a is an element of high multiplicative order modulo m (e.g., a primitive root modulo n), and the seed X_0 is coprime to m.
- In 1988, Park and Miller[3] suggested particular parameters $m = 2^{31} - 1 = 2,147,483,647$ (a Mersenne prime $M31$) and $a = 7^5 = 16,807$ (a primitive root modulo $M31$)

Park-Miller Random Number Generator

- We present a generator called Park & Miller - it provides a minimum guaranteed level of statistical accuracy.
- In the design, we present the generator in two pieces. A small inner class that develops a random generator that returns one integer every time it is called.
- A larger class that turns the output into a vector of uniforms in the format desired according to the designed interface.
- We call this pattern the *adaptor* pattern - where a class works and is effective, but its interface is not what the rest of the code expects.
- To use the adapter pattern simply means to use an intermediary class which transforms one interface into another.

The RandomBase Class

- Let's look at this new project with the RandomBase class:
  - Random2.h
  - Random2.cpp
  - ParkMiller.h
  - ParkMiller.cpp
- Note that we check whether the seed is *zero*. If it is we change it to 1. The reason is that a zero yields a chain of zeros.
- Note the advantage of a class-based implementation here. The seed is only inputted in the constructor and the set seed method, which are called only rarely, so we can put in extra tests to make sure the seed is correct with no real overhead.

Antithetic Sampling via Decoration

- A standard method of improving the convergence of Monte Carlo simulation is through antithetic sampling.

- It is easy to show that if $U \sim U(0,1)$, then

$$\mathbb{X} = \mathcal{F}^{-1}(U). \tag{7}$$

has cdf $\mathcal{F}$. Namely, since $\mathcal{F}$ is invertible and $\mathbb{P}(U \leq u) = u$, we have

$$\mathbb{P}(\mathbf{X} \leq x) = \mathbb{P}(\mathcal{F}^{-1}(U \leq x)) = \mathbb{P}(U \leq \mathcal{F}(x)) = \mathcal{F}(x). \tag{8}$$

Random Variable Generation

- **Inverse-Transform Method:** Let $\mathbf{X}$ be a random variable with cdf $\mathcal{F}$. Since $\mathcal{F}$ is a nondecreasing function, the inverse function $\mathcal{F}^{-1}$ may be defined as

$$\mathcal{F}^{-1}(y) = \inf\{x : \mathcal{F}(x) \geq y\}, 0 \leq y \leq 1. \tag{9}$$

- It is easy to show that if $U \sim U(0,1)$, then

$$\mathbb{X} = \mathcal{F}^{-1}(U). \tag{10}$$

has cdf $\mathcal{F}$. Namely, since $\mathcal{F}$ is invertible and $\mathbb{P}(U \leq u) = u$, we have

$$\mathbb{P}(\mathbf{X} \leq x) = \mathbb{P}(\mathcal{F}^{-1}(U \leq x)) = \mathbb{P}(U \leq \mathcal{F}(x)) = \mathcal{F}(x). \tag{11}$$

The Inverse-Transform Method

- **Algorithm (the Inverse-Transform Method)**

(1) *Generate U from U(0, 1)*

(2) *Return* $X = F^{-1}(U)$

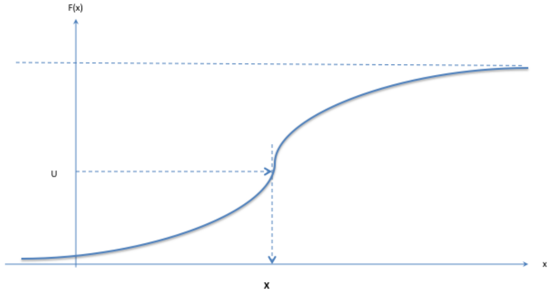


Figure: Inverse-Transform Method.

Example of the Inverse-Transform Method

Generate a random variable from the pdf

$$f(x) = \begin{cases} 2x, 0 \leq x \leq 1 \\ 0, \text{otherwise.} \end{cases} \tag{12}$$

The cdf is

$$F(x) = \begin{cases} 0, x < 0 \\ \int_0^x 2y dy = x^2, 0 \leq x \leq 1 \\ 1, x > 1 \end{cases} \tag{13}$$

Apply formula 10, we have

$$X = F^{-1}(U) = \sqrt{U}. \tag{14}$$

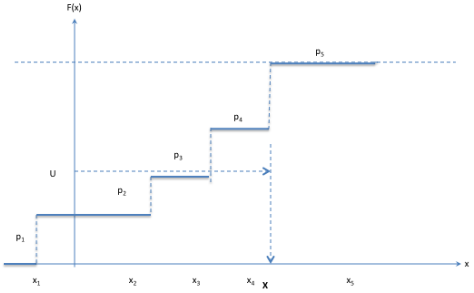
Therefore, to generate a random variable X from the pdf (13), first generate a random variable U from $U(0, 1)$ and then take its square root.

The Inverse-Transform Method for a Discrete Distribution

- **Algorithm (the Inverse-Transform Method)**

(a) *Generate U from U(0, 1)*

(b) *Find the smallest positive integer, k, such that $U \leq F(x_k)$ and return $X = x_k$.*



Example of the Inverse-Transform Method

Let X be a discrete random variable with $\mathbb{P}(\mathbf{X} = x_i) = p_i, i = 1, 2, ...,$ with $\sum_i p_i = 1$ and $x_1 < x_2 < ...$ The cdf $\mathcal{F}$ of $\mathbf{X}$ is given by $\mathcal{F}(x) = \sum_{i:x_i \leq x} p_i, i = 1, 2, ...$ and is illustrated in Figure (b).

- Much of the execution time in the above algorithm is spent in making the comparisons of Step 2. This time can be reduced by using efficient search techniques.
- In general, the inverse-transform method requires that the underlying cdf, F, exists in a form for which the corresponding inverse function $\mathcal{F}^{-1}$ can be found analytically or algorithmically. Applicable distributions are, for example, the exponential, uniform, Weibull, logistic, and Cauchy distributions.
- Unfortunately, many probability distributions can not be inverted or are not very efficient using the inverse-transform method.

Breakout Exercise: Sampling Comparison

- Complete the implementation of *StatisticsSTD* class to calculate variance of the statistics, and fit it into the class hierarchy given. Compare the variance of two pricing procedures: one with normal random numbers and another with antithetic random numbers.
- If you have more than one compiler see how they compare. Not that you can time things using the *clock*() function.
  - Pair with another person as a team; one is writing the code and another is watching and then take turns to complete the design.
  - Use *ch6_breakout_start* project as the base and add a new random number class.

Antithetic Sampling via Decoration

- A standard method of improving the simulation quality (variance reduction technique) of Monte Carlo simulations is antithetic sampling. The idea is very simple, if a X is a draw from a standard Gaussian distribution so is $-X$. This means that if we draw a vector $(X_1, ..., X_n)$ from one path then instead of drawing a new vector for the next path we simply use $(-X_1, ..., -X_n)$.
- This method guarantees that, for any even number of paths drawn, all the odd moments of the sample of Gaussian variates drawn are zero, and in particular the mean is correct. This generally, but not always, causes simulation to converge faster.
- We wish to implement antithetic sampling in such a way that it can be used with any random number generator and with any Monte Carlo simulation in such a way that we only have to implement it once.

Antithetic Sampling via Decoration

- Let's look at this new project with the VanillaOption class:
  - AntiThetic.h
  - AntiThetic.cpp
  - SimpleMC8.h
  - SimpleMC8.cpp
  - Random2.h
  - Random2.cpp
  - ParkMiller.h
  - ParkMiller.cpp
  - RandomMain3.cpp
- This program will compile and run but I do not like it. Why?
  - We create a Park-Miller random number generator object and then wrap it with an antithetic decorator. This decorator object then passed into the new Monte Carlo routine.
  - As usual, the routine is not aware of the fact that the passed-in object has been decorated but simply uses it in the same way as any other random number generator.

Key Points

- *rand* is implementation dependent.
- Results from *rand*() are not easily reproducible.
- We have to be sure that a random generator is capable of the dimensionality necessary for a simulation.
- Using a random number class allows us to use decoration.
- The inverse cumulative normal function is the most robust way to turn uniform variates from the open interval $(0, 1)$, into Gaussian variates.
- Using a random number class makes it easier to plug in low-discrepancy numbers.
- Antithetic sampling can be implemented via decoration.