BIJU PATNAIK UNIVERSITY OF TECHNOLOGY, ODISHA

Lecture Notes

On

Random Number Generation

Prepared by,

Dr. Subhendu Kumar Rath,

BPUT, Odisha.

Random Number Generation

Pseudo-random number

Generating Discrete R.V.

Generating Continuous R.V.

Pseudo-random numbers

- Almost random numbers!
- Actually, they are “computer-generated random numbers.”
- Not truly random because there is an inherent pattern in any sequence of pseudo-random numbers.
- Lot of mathematics goes into building pseudo-random number generators (algorithms) so that the output is sufficiently “random”.
- Most of these algorithms generate *U(0,1)* pseudo-random numbers.

Pseudo-random numbers

- John von Neumann (1951):

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin….There is no such thing as a random number – there are only methods to produce random number…We are dealing with mere 'cooking recipes' for making digits."

"These recipes …should be judged merely by their results. Some statistical study of the digits generated should be made, but exhaustive tests are impractical. If the digits work for one type of problem, they seem usually to be successful with others of the same type."

Random number generators

Properties of a “good” generator:

1. The number should appear to be distributed uniformly on [0,1] and should not exhibit any correlation.
2. Fast and avoid need for lot of storage.
3. Able to reproduce a given stream of random numbers exactly. Why?
4. Provision for producing several separate streams of random numbers.

Linear Congruential Generators (LCG)

- Let X_0 be the seed value (initial value of the sequence).
- Then the next values are generated by:

$$X_n = (aX_{n-1} + c)(\text{mod } m).$$

- *a* = the multiplier; *c* = increment; and *m* = modulus.
- Obviously, X_n can take values: *0, 1, … m-1*.
- The *U(0,1)* pseudo-random number is generated by X_n/m.
- Values of *a*, *m*, and *c* are chosen such that LCG satisfies the properties of "good" generator.

Linear Congruential Generators (LCG)

- For a 32-bit computer, typical values of the parameters are: $a = 7^5 = 16{,}807$ and $m = 2^{31} - 1$. c should be such that the only positive integer that (exactly divides) both c and m is 1.
- LCG is bound to "loop."
- Whenever, X_i takes on a value it had previously taken, exactly the same sequence of values is generated and cycle repeats.
- The length of the cycle is called *period* of a generator.
- Length of the period can almost be m.
- If in fact the period is m, then it is called *full period LCG*.
- If the period is less than m, the cycle length depends on X_0.

Testing random number generators

Uniformity test

- Check whether the values are uniformly distributed between *0* and *1*.
- As before, we employ a chi-square test statistic to compare the observed frequency and expected frequency of observations in an interval.
- We create *k* intervals and count the number of times the observation falls in a particular interval. If the r.v. generated are *U(0,1)*, the expected frequency should be same for each (equal-sized) interval.

Testing random number generators

Serial test

- Generalization of the chi-square test in higher dimensions. If individual values are uniformly distributed, then the vector in *d*-dimensions should be uniformly distributed over a hyper-cube $[0,1]^d$.
- Alternately, if individual values are correlated, the distribution of the *d*-vectors will deviate from *d*-dimensional uniformity.
- Thus, serial test indirectly tests for independence of individual observations.

Testing random number generators

Runs test

- Directly tests for independence.
- We examine the sequence of values for unbroken subsequence of maximal length within which the U_i's grow monotonically. Such a sequence is called *run up*.

$$r_i = \begin{cases} \text{number of runs up to length } i & i = 1,2,\ldots 5 \\ \text{number of runs up to length} \geq 6 & i = 6. \end{cases}$$

Testing random number generators

- The test statistic is calculated as:

$$R = \frac{1}{n} \sum_{i=1}^{6} \sum_{j=1}^{6} a_{ij} (r_i - nb_i)(r_j - nb_j).$$

where *A* and *b* are given matrices.

- Turns out that the test statistic *R* has chi-square distribution.
- So this calculated value can be compared with the tabular value to check for the hypothesis of independence.

Generating random variates

- Activity of obtaining an observation on (or a realization of) a random variable from desired distribution.
- These distributions are specified as a result of activities discussed in last chapter.
- Here, we assume that the distributions have been specified; now the question is how to generate random variates with this distributions to run the simulation experiment.
- The basic ingredient needed for *every* method of generating random variates from *any* distribution is a source of IID *U(0,1)* random variates.
- Hence, it is essential that a statistically reliable *U(0,1)* random number generator be available.

Requirements from a method

Exactness

- As far as possible use methods that results in random variates with exactly the desired distribution.
- Many approximate techniques are available, which should get second priority.
- One may argue that the fitted distributions are approximate anyways, so an approximate generation method should suffice. But still exact methods should be preferred.
- Because of huge computational resources, many exact and efficient algorithms exist.

Requirements from a method

Efficiency

- Efficiency of the algorithm (method) in terms of storage space and execution time.
- Execution time has two components: set-up time and marginal execution time.
- Set-up time is the time required to do some initial computing to specify constants or tables that depend on the particular distribution and parameters.
- Marginal execution time is the incremental time required to generate each random variate.
- Since in a simulation experiment, we typically generate thousands of random variates, marginal execution time is far more than the set-up time.

Requirements from a method

Complexity

- Of the conceptual as well as implementational factors.
- One must ask whether the potential gain in efficiency that might be experienced by using a more complicated algorithm is worth the extra effort to understand and implement it.
- “Purpose” should be put in context: a more efficient but more complex algorithm might be appropriate for use in permanent software but not for a “one-time” simulation model.

Robustness

- When an algorithm is efficient for all parameter values.

Inverse transformation method

- We wish to generate a random variate X that is continuous and has a distribution function that is continuous and strictly increasing when $0 < F(x) < 1$.
- Let F^{-1} denote the inverse of the function F.
- Then the inverse transformation algorithm is:

1. Generate $U \sim U(0,1)$
2. Return $X = F^{-1}(U)$.

- To show that the returned value X has the desired distribution F, we must show $\Pr\{X \le x\} = F(x)$. How?

Inverse transformation method

- This method can be used when *X* is discrete too. Here,

$$F(x) = \Pr\{X \le x\} = \sum_{x_i \le x} p(x_i).$$

where, $p(x_i) = Pr\{X = x_i\}$ is the probability mass function.

- We assume that *X* can take only the values $x_1, x_2, \ldots$ such that $x_1 < x_2 < \ldots$
- The algorithm then is:

1. Generate $U \sim U(0,1)$.
2. Determine the smallest integer *I* such that $U \le F(x_I)$, and return $X = x_I$.

Inverse transformation method

Advantages:

- Intuitively easy to understand.
- Helps in variance reduction.
- Helps in generating rank order statistics.
- Helps in generating random variates from truncated distributions.

Disadvantages:

- Closed form expression for F^{-1} may not be readily available for all distributions.
- May not be the fastest and the most efficient way of generating random variates.

Composition

- Applicable when the desired distribution function can be expressed as a convex combination of several distribution functions.

$$F(x) = \sum_{j=1}^{\infty} p_j F_j(x),$$

where $p_j \geq 0, \sum_{j=1}^{\infty} p_j = 1$; and each F_j is a distribution function.

The general composition algorithm is:

1. Generate a positive random integer J such that:

   $$\Pr\{J = j\} = p_j,\ j = 1,2,\ldots$$

2. Return X with distribution function F_j.

Acceptance-Rejection technique

- All the previous methods were direct methods – they dealt directly with the desired distribution function.
- This method is a bit indirect.
- Applicable to continuous as well as discrete case.
- We need to specify a function *t* such that $f(x) \le t(x)\ \ \forall\, x.$ We say that *t majorizes* density *f*. In general, function t will not be a density function, because:

$$c = \int_{-\infty}^{\infty} t(x)dx \ge \int_{-\infty}^{\infty} f(x)dx = 1.$$

- However, the function *r(x) = t(x)/c* clearly will be a density.

Acceptance-Rejection technique

- We should be able to (easily) generate a random variate *Y* having density *r* (using one of previous methods).

The acceptance-rejection algorithm is:

1. Generate *Y* having density *r*.
2. Generate *U* ~ *U(0,1)* independently of *Y*.
3. If $U \le \frac{f(Y)}{t(Y)}$ return $X = Y$.

   Otherwise, go back to Step *1* and try again.

Generating continuous random variates

Uniform (a,b)

Using inverse-transform method

1. Generate $U \sim U(0,1)$.
2. Return $X = a + (b-a)\,U$.

Exponential (β)

Once again, using inverse transform method

1. Generate $U \sim U(0,1)$.
2. Return $X = -\,\beta\,ln(U)$.

Generating continuous random variates

Normal – *N(0,1)*

- Earliest method from 1958 (still very popular)

1. Generate U_1, U_2 as IID U(0,1).
2. Set

$$X_1 = \cos(2\pi U_2)\sqrt{-2\ln(U_1)}$$

$$X_2 = \sin(2\pi U_2)\sqrt{-2\ln(U_1)}$$

$$X_1, X_2 \sim N(0,1).$$

- Drawback: If U_1 *and* U_2 are generated using adjacent random numbers from LCG, then X_1, X_2 are not truly independent.

Generating continuous random variates

Normal – *N(0,1)*

Polar method

1. Generate U_1, U_2 as IID $U(0,1)$. $V_i = 2U_i - 1,\ i = 1,2.$

   $W = V_1^2 + V_2^2.$

2. If $W > 1$, go back to Step *1*. Otherwise let

$$Y = \sqrt{\frac{-2\ln(W)}{W}};$$

$$X_1 = V_1 Y;\quad X_2 = V_2 Y. \qquad X_1, X_2 \sim N(0,1)\ \text{IID}.$$

Generating random variates from empirical distribution

- For empirical distribution from exact data, recall that:

$$F(x) = \begin{cases} 0 & if \quad x < X_{(1)} \\ \frac{i-1}{n-1} + \frac{x - X_{(i)}}{(n-1)\left(X_{(i+1)} - X_{(i)}\right)} & if \;\; X_{(i+1)} \le x < X_{(i)}, i = 1,2,\ldots n-1 \\ 1 & if \;\; X_{(n)} \le x. \end{cases}$$

- Algorithm:

1. Generate U~U(0,1)
2. Let P = (n-1)U and let $I = \lfloor P \rfloor + 1.$
3. Return: $X = X_{(I)} + (P - I + 1)\left(X_{(I+1)} - X_{(I)}\right).$

Generating random variates from empirical distribution

For the grouped data.

1. Generate $U \sim U(0,1)$.
2. Find the non-negative integer J $(0 \leq J \leq k-1)$ such that
   $G(a_j) \leq U \leq G(a_{j+1})$.

3. Return:

$$X = a_j + \frac{\lfloor U - G(a_j) \rfloor (a_{j+1} - a_j)}{(G(a_{j+1}) - G(a_j))}.$$

Generating discrete random variates

Bernoulli *(p)*

1. Generate $U \sim U(0,1)$.
2. If $U \leq p$, return $X = 1$; otherwise $X = 0$.

Is this a inverse transform method?

Binomial *(n,p)*

1. Generate $Y_1, Y_2, \ldots, Y_n$ as IID *Bernoulli (p)* random variates.
2. Return $X = Y_1 + Y_2 + \ldots + Y_n$.

Generating discrete random variates

Discrete Uniform *(i, j)*

1. Generate $U \sim U(0,1)$.

2. Return $X = i + \lfloor (j - i + 1)U \rfloor$.

No search is required. We can compute and store $(j\text{-}i+1)$ ahead of time to reduce computation time.

This is exactly the inverse transform method, too!

Generating discrete random variates

Poisson (λ)

1. Let $a = e^{-\lambda}$, $b = 1$ and $i = 0$.
2. Generate $U_{i+1} \sim U(0,1)$ and replace b with bU_{i+1}. If $b < a$, then $X = i$. Otherwise go to step 3.
3. Replace i with $i + 1$ and go back to step 2.