13 Random Variables and Simulation

In this lecture note we consider the relationship between random variables and simulation models. Random variables play two important roles in simulation models.

We assume that within our models some delays (`pause()` or `process()` in SimJava2) will not have deterministic values, but instead will be represented by random variables; similarly when a choice must be made within the behaviour of an entity we will sometimes want the decision to be made probabilistically. Both cases will involve sampling a probability distribution to extract a value each time this part of the entity's behaviour is reached. For delays we will use continuous random variables (e.g. `Sim_normal_obj` in SimJava2) and for a choice we will use a boolean random variable (via `Sim_random_obj` in SimJava2). As we will see below, both cases rely on the random number generator.

As with the models we have considered earlier in the course, we assume that the variables characterising the behaviour of the system/model, the performance measures or output parameters, are also random variables. For example, a measure might be the number of packets lost per million in a communication network, or the average waiting time for an access to a disk. In general, each run of the simulation model provides a single estimate for these random variables. If we are interested in steady state values the longer we run a simulation the better our estimate will be. However, it still remains a single observation in the sample space. We need more than a single estimate in order to draw conclusions about the system. Thus we use output analysis techniques to improve the quality of an estimate from a single model run, and to develop ways of gaining more observations without excessive computational cost. Realistic simulation models take a long time to run—there is always a trade-off between accuracy of estimates and execution time.

13.1 Random Number Generation

Generating random values for variables with a specified random distribution, such as an exponential or normal distribution, involves two steps. First, a sequence of random numbers distributed uniformly between 0 and 1 is obtained. Then the sequence is transformed to produce a sequence of random values which satisfy the desired distribution. This second step is sometimes called *random variate generation*.

To obtain a sequence of uniform random numbers, it is sufficient to be able to generate a sequence X_k of integers in the range $[0, M-1]$ since the sequence $X_k/(M-1)$ will then be approximately uniformly distributed over $(0, 1)$. In 1951, D.H. Lehmer discovered that the residues of successive powers of a number have good randomness properties. He obtained the kth number in the sequence by dividing the kth power of an integer a by another integer M and taking the remainder.

$$X_k = a^k \bmod M$$

This can be expressed as an iteration:

$$X_k = (a \times X_{k-1}) \bmod M$$

The parameters a and M are called the *multiplier* and the *modulus* respectively. Random number generators of this form are called *Lehmer generators*, or *multiplicative linear-congruential generators*.

There are several desirable properties for a random number generator:

It should be efficiently computable. Since simulations typically require several thousand random numbers in each run, the processor time required to generate these numbers should be small.

It should be pseudo-random. Given the same seed the random number generator should produce exactly the same sequence of numbers. Without this property it would not be possible to recreate experiments.

The cycle should be long. A short cycle may cause the random number sequence to recycle, resulting in a repeated event sequence. This may limit the useful length of simulation runs.

Successive values should be independent and uniformly distributed. The correlation between successive numbers should be small. Correlation, if significant, indicates dependence.

Research has shown that Lehmer generators obey these properties provided a and M are carefully chosen. However care is needed. In the early 1970s most university mainframes were using a linear-congruence generator known as `RANDU`. It used the values $a = 65539$ and $M = 2^{31}$. Although the output looked random, detailed statistical analysis showed that there was significant correlation in the output.

Nevertheless this form of generator continues to be used, if somewhat more warily. These generators are particularly efficient if M is chosen to be a power of 2. In this case finding the residue amounts to simply truncating the result of the multiplication. However a modulus of the form 2^k results in a shorter cycle: 2^{k-2} at best.

Care must be taken in the implementation of random number generators. For example, properties such as the length of cycle are only maintained if all computations are done precisely without round-off errors. Thus the computations should use only integer arithmetic. Similarly, care must be taken to ensure that the multiplication $a \times X_{k-1}$ does not cause overflow—intermediate calculations may need to be carried out using extended precision arithmetic operations.

The random number generator used in SimJava2 is a Lehmer generator with parameters: $a = 742,938,285$ and $M = 2^{31} - 1$. This has been demonstrated to have good statistical properties.

Generation algorithms for values of the commonly used probability distributions, based on a uniformly distributed stream of values between 0 and 1, can be found in many books on simulation and performance modelling[1]. Inverse transformation algorithms are based on the observation that for any probability distribution with distribution function $F(x)$, the value of $F(x)$ is uniformly distributed between 0 and 1. Thus, using values from the random number stream, $u = X_k$, the function is inverted to find the next value of x: $x = F^{-1}(u)$. For example, given a random number u, we generate the next value in an

[1]For example, *The Art of Computer Systems Performance Analysis*, R. Jain, Wiley, 1991.

exponential distribution with parameter λ as[2]

$$x = -\frac{1}{\lambda}\ln(u)$$

Boolean valued distributions which are used to make decisions within a model take a single real parameter, p, such that $0 \leq p \leq 1$. This represents the probability of a "positive" outcome. Then each time the branching point in the model is reached, the next random number in the stream is generated $u = X_k$. If $u \leq p$ the positive branch is taken; if $u > p$ the other branch is selected.

Similarly, for discrete probability distributions a technique known as *aliasing* is used. Here a cumulative distribution function is calculated based on the probability mass function, once some order is imposed on possible outcomes. Random values between 0 and 1 are then compared with the cumulative function, and the appropriate outcome selected based on the interval into which the value falls.

One of the benefits of using a simulation package is that at least some of these algorithms are provided for us. SimJava2 provides classes for many commonly used continuous probability distributions: e.g. `Sim_normal_obj, Sim_negexp_obj`, and `Sim_uniform_obj` and discrete probability distributions: e.g. `Sim_bernoulli_obj`, `Sim_binomial_obj`, and `Sim_poisson_obj`. Each time such an object is instantiated the seed for the random number generator can be set explicitly. However if these seeds are not *well-spaced* there may be overlap between the sequences of random numbers used by the generators resulting in correlation between the samples used in the simulation. Consequently SimJava2 also provides an automatic seeding mechanism which will seed each distribution with a distinct seed which is far in the cycle from other seeds currently in use. This avoids the problem of overlap and correlation.

13.2 Simulation output analysis

In performance modelling our objective in constructing a simulation model of a system is to generate one or more performance measures for the system. In the Markov models we considered earlier in the course such measures were derived from the steady state probability distribution, after the model had been solved. In contrast, in a simulation model measures are evaluated directly during the execution of the model. It is part of model construction to make sure that all the necessary counters and updates are in place to allow the measures to be collected as the model runs. This is sometimes called *instrumentation* of a model as it is analogous to inserting probes and monitors on a real system.

However the data from a simulation model is generated it is important to remember that each run of a model constitutes a single trajectory over the state space. Consequently, in general, any estimate for the value of a performance measure generated from a single run constitutes a single observation in the possible sample space. To gain an accurate measure of the performance of the system we should not base our results on a single observation.

[2]Strictly speaking, the equation should be $x = -1/\lambda \ln(1-u)$ but since u is uniformly distributed between 0 and 1, $1-u$ will be uniformly distributed between 0 and 1 and the generation algorithm can be simplified.

When carrying out steady state analysis we should bear in mind that the averages we calculate from data collected during execution will always be an approximation of the unknown true long-term averages that characterise the system performance. Important issues are:

- choosing the starting state of the simulation;
- choosing the *warm-up* period that is allowed to elapse before data collection begins;
- choosing a run length that ensures that the calculated averages are representative of the unknown true long term average.

Statistical techniques can be used to assess how and when the calculated averages approximate the true average, i.e. to analyse the accuracy of our current estimate. This is often done in terms of a *confidence interval*. A confidence interval expresses probabilistic bounds on the error of our current estimate. So a confidence interval (c_1, c_2) with *confidence level* $X\%$, means that with probability $X/100$ the real value v lies between the values c_1 and c_2, i.e.

$$\Pr(c_1 \leq v \leq c_2) = X/100$$

$X/100$ is usually written in the form $1 - \alpha$, and α is called the *significance level*, and $(1 - \alpha)$ is called the *confidence coefficient*. Usually performance modellers will run their simulation models until their observations give them confidence levels of 90% or 95% and a confidence interval which is acceptably tight. Calculation of the confidence interval is based on the variance within the observations which have been gathered. The greater the variance the wider the confidence interval; the smaller the variance the tighter the bounds.

In SimJava2 the confidence level can be used to control the period over which a model is run or the number of times it is run for, as we will see later in this note.

13.2.1 Steady state measures

For some modelling studies the length of time for which a simulation model should be run is defined by the problem itself. For example, if we wish to investigate how many messages can be processed by a dealers' transaction processing system *in the first hour of trading* then it makes sense to run the model for 3600 seconds. However, if the question is how many messages can be processed by the dealers' transaction processing system *in an average hour* then running the model for 3600 seconds is unlikely to be enough.

The first question identifies the simulation as a *transient* or *terminating* simulation. It is said to have a *cold-start*: the system is initially empty which is not its usual state but we still include this data in the observation period. For this type of simulation the question becomes how many times the simulation must be repeated (with different random number streams) to achieve a required confidence interval.

In the second scenario we are interested in the *steady state* behaviour of the system. As in Markovian modelling we associate steady state behaviour with long term behaviour. In other words we are theoretically interested in the observations obtained from runs of the model which are infinitely long. However, in practice we are interested in finite run

lengths and estimating the steady state distribution of the measures we are interested in from finitely many samples.

The initial conditions of the model, its *starting state*, influence the sequence of states through which the simulation will pass, especially near the start of a run. In a steady state distribution the output values should be independent of the starting state. Thus the modeller must make some effort to remove the effect of the starting state, sometimes termed *bias*, from the sample data used for estimating the performance measure of interest. Unfortunately it is not possible to define exactly when the model has moved from transient behaviour to steady state behaviour. This initial period before steady state is reached is sometimes called the *warm-up period*. Therefore, although several techniques exist, they are all heuristics. The common techniques are

1. Long runs.
2. Proper initialisation.
3. Truncation.
4. Initial data deletion.
5. Moving average of independent replications.
6. Batch means.

The last four techniques are all based on the assumption that variability is less during steady state behaviour than during transient behaviour.

In SimJava2 the end of warm-up period can be set using a condition: steady state measures will only be calculated based on data collected after the condition became true. The method `Sim_system.set_transient_condition()` is used to set the condition to be used. There are three possibilities:

- The condition is expressed in terms of event completions based on an event tag at a given entity, e.g. `Sim_system.set_transient_condition(Sim_system.EVENTS_COMPLETED, "Server", 0, 5);` in the `AppletQ_Stats` example marks the end of the warm-up period as when 5 jobs have been served at the server.
- An estimate the length of the warm-up period is used to express the condition as a simulation time, e.g. `Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);` in `Queue_Other` SimJava2 will start collecting data when 50000 time units have elapsed.
- No condition is set by the modeller but after the completion of a run SimJava2 applies the minimum-maximum truncation method to identify the end of the transient period. Note however that this is a crude technique which generally under-estimates the length of the warm-up period.

13.2.2 Termination Conditions

How the simulation run is terminated can also have an effect on the accuracy of the data collected. There are two generally used options for choosing when to stop the simulation of a performance model:

Option 1

- begin the simulation at time 0
- begin data collection at specified time $w \geq 0$
- complete data collection at specified time $w + t$
- terminate execution of the simulation at time $w + t$
- calculate summary statistics based on sample path data collected in the time interval $(w, w + t)$.

Option 2

- begin the simulation at time 0
- begin data collection when the Mth job completes all service
- complete data collection when the $(M + N)$th job completes all service
- terminate execution of the simulation when the $(M + N)$th job completes all service
- calculate summary statistics based on sample path data collected in the time interval (t_M, t_{M+N}), where t_j is the time at which the jth job completes all service.

Option 1 implies that the simulated time $(w, w + t)$ for data collection is predetermined but the number of job completions is random. Conversely, Option 2 implies that the time period for data collection is random but the number of job completions is predetermined. Option 1 is preferable for calculating queue lengths and resource utilisations, whereas Option 2 is preferable for calculating waiting times.

Both options are supported in SimJava2, and as with the transient condition, the termination condition can be based on either event completions or elapsed time. For example in the example model `Queue.java` the simulation is set to terminate when 10 jobs have been processed at the Server:

```
Sim_system.set_termination_condition(Sim_system.EVENTS_COMPLETED,
                                                    "Service", 0, 10, false);
```

A third option is to use confidence interval accuracy as the controlling factor.

13.2.3 Variance reduction techniques

Assume that we are running a simulation model in order to estimate some performance measure M. During the ith execution of the model we make observations of M, o_{ij} and at the end of the run we calculate the mean value of the observations O_i. Note that the observations o_{ij} in most simulations are *not* independent. Successive observations are often correlated. For example, if we are interested in the delay of messages in a packet-switching network, if the delay of one message is long because the network is heavily congested, the next message is likely to be similarly delayed. Thus the two observations are not independent. This is why, in general, a simulation model must be run several times.

Independent Replications If independent replications are used the model is run m times in order to generate m independent observations. In order for the runs to be independent the seeds used for the random number generator must be carefully chosen to ensure that they are independent.

If transient or short term behaviour is being investigated all the data collected during the ith run will be used to calculate the ith observation, i.e. O_i is the mean value over all o_{ij}.

If steady state or long term behaviour is being investigated the data relating to the warm-up period must be discarded. The observation for the run, O_i will be the mean value over o_{ij} such that $j > k$, where $\{o_{i1}, \ldots o_{ik}\}$ are the observations made during the warm-up period.

In either case let O denote the mean value of the observations, O_i, after m runs. Then the variance over all observations is calculated as shown below:

$$V = \frac{1}{m-1} \sum_{i=1}^{m} (O_i - O)^2$$

For steady state analysis independent replication is an inefficient way to generate samples, since for each sample point, O_i, k observations, $\{o_{i1}, \ldots, o_{ik}\}$, must be discarded.

Batch Means In the method of batch means the model is run only once but for a considerable period. The run is divided into a series of sub-periods of length ℓ, and measures are collected over each sub-run to form a single point estimate. If the observations made during the run form a set $\{o_i\}$, the set is partitioned into subsets

$$S_i = \{o_j \mid o_j \text{ observed between } (i-1) \times \ell \text{ and } i \times \ell\}$$

Now each sample point O_i is the mean generated from a subset of observations S_i, and O is the mean generated from the O_i. Variance is calculated as above.

This method is unreliable since the sub-periods are clearly not independent. However it has the advantage that only one set of observations $\{o_i \ldots o_k\}$ needs to be discarded to overcome the warm-up effects in steady state analysis.

Regeneration It is sometimes possible within the run of a simulation model to identify points in the trajectory where the model returns to exactly equivalent states. These are called *regeneration points*. Periods between regeneration points are genuinely independent sub-runs. The simplest example to consider is the state when a queue becomes empty. Whatever the distribution used to generate the behaviour of the model the trajectory (queue length, waiting time etc) after a visit to this state does not depend on the previous history of the model in any way. The duration between two successive regeneration points is called a *regeneration cycle*.

The variance computation using regeneration cycles is a bit more complex than that in the method of batch means or the method of independent replications. This is because the regeneration cycles are of different lengths, whereas in the other two methods the batches or replications are all of the same length. Suppose that we divide a sample into m cycles of length $n_1, \ldots, n_m$ respectively. The mean for each cycle, generating a single

observation sample can be calculated as expected: $O_i = \frac{1}{n_i} \sum_{j=1}^{n_i} o_{ij}$. However the overall mean O, is not the arithmetic mean of the O_i. Instead we use a different approach to calculate O:

$$O = \frac{\sum_{i=1}^{m} \sum_{j=1}^{n_i} o_{ij}}{\sum_{i=1}^{m} n_i}$$

Variance of the summed observations is then calculated as shown below:

$$V_s = \frac{1}{m-1} \sum_{i=1}^{m} (n_i(O_i - O))^2$$

Notice that, unlike the previous two methods, the method of regeneration does not require any transient observations to be removed. Unfortunately not all models have easily defined regeneration states, and even when they exist they can be computationally expensive to identify. Another disadvantage is that it is not possible to define the length of a simulation run beforehand. However, research suggests that this method gives the most accurate results and so it is the method of choice when it is feasible.

SimJava2 supports both independent replications and batch means and use of these can be specified via the `Sim_system.set_output_analysis()` method. If the method is not used no output analysis will be performed. The method call allows the modeller to set the technique to be used and its parameters, such as the number of replications and the desired confidence level for confidence intervals. The output analysis is carried out automatically. For example, a call

```
Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
```

will run the model for 5 independent replications and confidence level 95%. At the completion of each run the model will be reset to its initial starting state and all generators will be re-seeded. If an entity contains mutable objects or non-final static fields the modeller must include code in the `body()` method which will reset them as this will not be handled automatically (see the SimJava2 tutorial for more details).

Variance reduction can also be used as a termination condition. For example in the model `Queue_Variance_Reduction.java` a confidence level is set at 95% and batch means analysis is used to control the length of the run until this confidence level is achieved.

```
Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);
Sim_system.set_termination_condition(Sim_system.INTERVAL_ACCURACY,
       Sim_system.BATCH_MEANS, 0.95, 0.1, "Service", Sim_stat.UTILISATION);
```