3 Discrete-Event Simulation

Roughly speaking, there are three different kinds of systems: continuous systems, discrete systems and discrete-event systems. A continuous system is a system which state varies continuously in time. Such systems are usually described by a set of differential equations. For example, continuous systems often arise in chemical applications. A discrete system is a system which is observed only at some fixed regular time points. Such systems are usually described by a set of difference equations. An example of a discrete system is an inventory model in which we inspect the stock only once a week. The characteristic feature of a discrete-event system is that the system is completely determined by a sequence of random event times $t_1, t_2, \ldots$, and by the changes in the state of the system which take place at these moments. Between these event times the state of the system may also change, however, according to some deterministic pattern. A first example of a discrete-event simulation was given in the previous section when we considered the two-machine production line. A second example of a discrete-event system is a single server queueing system.

A way to simulate a continuous system or a discrete system is by looking at the model at regular time points $0, t, 2t, \ldots$. Events which occur somewhere in the interval between two of these points are taken to happen at one of the end points of the interval. This approach is called synchronous simulation. For continuous systems it might be necessary to take $t$ very small in order to obtain a sufficiently accurate simulation and hence the simulation may be very slow.

For a discrete-event system it is not efficient to use a synchronous simulation. Instead, we use an asynchronous simulation, i.e. we jump from one event time to the next and describe the changes in the state at these moments. In this course, we will concentrate on asynchronous simulations of discrete-event systems.

When you write a simulation program for a discrete-event system you can take an event scheduling approach or a process-interaction approach. The event-scheduling approach concentrates on the events and how they affect system state. The process-interaction approach concentrates on the different entities in the system (e.g. the customers, the server and the arrival generator) and describes the sequence of events and activities such entities execute or undergo during their stay in the system. When using a general-purpose language, such as Fortran, Pascal or C, one uses in general the event-scheduling approach. However, simulation languages as Simscript or GPSS use the process-interaction approach. In this course, we will concentrate on the event-scheduling approach.

As an example of a discrete-event simulation we will consider the $G/G/1$ queueing system. Suppose that we use a simulation study to obtain an estimator for the long-term average waiting time of customers in this queueing system.

First, we remark that we can simulate the waiting time in a $G/G/1$ queue by using a simple discrete simulation instead of a discrete-event simulation. If we denote by $W_n$ the waiting time of the $n$-th customer, by $B_n$ the service time of the $n$-th customer and by $A_n$ the interarrival time between the $n$-th and the $(n+1)$-st customer, then we can obtain the following difference equation for $W_n$:

$$W_{n+1} = \max(W_n + B_n - A_n, 0).$$

Hence, an estimator for the long-term average waiting time of customers can be obtained by the following discrete simulation program:
PROGRAM: G/G/1 QUEUEING SYSTEM
(discrete simulation)

INITIALIZATION

n := 0;
w := 0;
sum := 0;

MAIN PROGRAM

while n < maximum
do
    a := new interarrival time;
b := new service time;
w := max(w+b-a,0);
sum := sum + w;
n := n + 1;
end

OUTPUT

mean waiting time := sum / n ;

However, if we want to obtain more information from the simulation (e.g. the queue length distribution or the fraction of time the server is busy) then a discrete simulation would not be suitable. Also, if we consider more complicated queueing systems a simple recursive formula is not available and hence a discrete simulation would not work anymore. Hence, we will also describe a discrete-event simulation for the long-term waiting time in the $G/G/1$ queue.

The event times are the arrival and departure moments of customers. A difference with the example of the production line is that in the $G/G/1$ case we do not know beforehand in what sequence the different events will take place. Whether the next event is an arrival event or a departure event depends on the realizations of the interarrival times and service times of the customers. Therefore, we will make a so-called event list in which we list the next arrival event and the next departure event and we jump to the event which will occur the first.

Let us look in some detail what happens at the different event times for the $G/G/1$ queueing system. At an arrival event, we certainly have to generate a new interarrival time. Furthermore, if the server is free, we start a new service, i.e. we generate a new service time. On the other hand, if the server is busy, we add the customer to the queue. At a departure event, we remove the leaving customer from the system. Furthermore, if the queue is not empty, we start a new service
and register how long the customer was in the queue. This leads to the following program:

<table>
<thead>
<tr>
<th>PROGRAM: G/G/1 QUEUEING SYSTEM (discrete-event simulation)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INITIALIZATION</strong></td>
</tr>
<tr>
<td>time := 0;</td>
</tr>
<tr>
<td>queue := nil;</td>
</tr>
<tr>
<td>sum := 0; throughput := 0;</td>
</tr>
<tr>
<td>generate first interarrival time;</td>
</tr>
<tr>
<td><strong>MAIN PROGRAM</strong></td>
</tr>
<tr>
<td>while time &lt; runlength</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>case nextevent of</td>
</tr>
<tr>
<td>arrival event:</td>
</tr>
<tr>
<td>time := arrivaltime;</td>
</tr>
<tr>
<td>add customer to queue;</td>
</tr>
<tr>
<td>start new service if server is idle;</td>
</tr>
<tr>
<td>generate new interarrival time;</td>
</tr>
<tr>
<td>departure event:</td>
</tr>
<tr>
<td>time := departuretime;</td>
</tr>
<tr>
<td>throughput := throughput + 1;</td>
</tr>
<tr>
<td>remove customer from queue;</td>
</tr>
<tr>
<td>if queue not empty</td>
</tr>
<tr>
<td>sum := sum + waiting time;</td>
</tr>
<tr>
<td>start new service;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td><strong>OUTPUT</strong></td>
</tr>
<tr>
<td>mean waiting time := sum / throughput;</td>
</tr>
</tbody>
</table>

Remark that, in order to be able to calculate the waiting time of a customer, we must remember the arrival time of a customer at the moment that the customer gets into service. Furthermore, we must be able to add and remove customers from a queue. In principal, this can be done by using an array as data structure. However, we prefer to use records and linked lists. Each customer is represented by a record consisting of his arrival time and a reference to the customer who is next in the queue.

In general, in an event-oriented approach of a discrete-event simulation there are a number of, say, \( N \) different events, each with its own corresponding activities \( A_1, \ldots, A_N \). In the event list
we keep track of the time points $T_1, \ldots, T_N$ at which the next events of the different types occur. The simulation then consists of finding the smallest $T_i$, setting the current time to this event time and executing the corresponding activities. A typical construction of a discrete-event simulation program is:

```
DISCRETE-EVENT SIMULATION

INITIALIZATION
set simulation time equal to 0;
initialize system state and statistical counters;
initialize event list;

MAIN PROGRAM
while simulation time < runlength
do
determine next event type;
advance simulation time;
update system state + statistical counters;
generate future events + add them to event list;
end

OUTPUT
compute + print estimates of interest;
```

3.1 Verification of a simulation model

One important issue during a simulation study is the following: How do we check whether a program of a discrete event simulation model is operating as intended? One of the techniques to do this is by using a trace. In a trace, the state of the simulated system, i.e. the values of the state
variables, the statistical counters and the contents of the event list, is printed just after each event occurs. For a small runlength the results of the trace can then be compared with calculations by hand.

Furthermore, a model should, when possible, be run under special assumptions for which results can also be obtained by analysis. For example, we can first simulate the production line of section 2 with zero and infinite buffer. Also, we can simulate the system with exponential lifetimes and repair times to see whether or not the result of the simulation coincides with the result of the analysis. For the $G/G/1$ system we can first simulate the system with exponential interarrival and service times, for which it is possible to obtain exact results (see e.g. the courses on Stochastic Processes (2S500) and Queueing Theory (2S520)).

Finally, it is of course sensible to write and check a simulation program in modules or subprograms.
4 Random-Number Generators

A simulation of a system which contains random components requires a method of generating random numbers. This section is devoted to the generation of random numbers from the uniform (0, 1) distribution. Although this is only one of all possible distribution functions, it is very important to have an efficient way of generating uniformly distributed random numbers. This is due to the fact that random variables from other distributions can often be obtained by transforming uniformly distributed random variables. We come back to this issue in the next section.

A good random number generator should satisfy the following properties:

1. The generated numbers should satisfy the property of uniformity.
2. The generated numbers should satisfy the property of independence.
3. The random numbers should be replicable.
4. It should take a long time before numbers start to repeat.
5. The routine should be fast.
6. The routine should not require a lot of storage.

The generation of random numbers has a long history. In the beginning, random numbers were generated by hand using methods as throwing dice or drawing numbered balls. However, nowadays only numerical ways to generate random numbers are used. Usually, the random numbers are generated in a sequence, each number being completely determined by one or several of its predecessors. Hence, these generators are often called pseudo-random number generators.

4.1 Pseudo-Random Number Generators

Most pseudo-random number generators can be characterized by a five-tuple \((S, s_0, f, U, g)\), where \(S\) is a finite set of states, \(s_0 \in S\) is the initial state, \(f : S \rightarrow S\) is the transition function, \(U\) is a finite set of output values, and \(g : S \rightarrow U\) is the output function.

The pseudo random-number generator then operates as follows:

- (1) Let \(u_0 = g(s_0)\).
- (2) For \(i = 1, 2, \ldots\) let \(s_i = f(s_{i-1})\) and \(u_i = g(s_i)\).

The sequence \((u_0, u_1, \ldots)\) is the output of the generator. The choice of a fixed initial state \(s_0\) rather than a probabilistic one makes replication of the random numbers possible.

Let us next give a number of examples of pseudo-random number generators.
4.1.1 Midsquare Method

One of the first pseudo-random number generators was the midsquare method. It starts with a fixed initial number, say of 4 numbers, called the seed. This number is squared and the middle digits of this square become the second number. The middle digits of this second number are then squared again to generate the third number and so on. Finally, we get realizations from the uniform (0,1) distribution after placement of the decimal point (i.e. after division by 10000). The choice of the seed is very important as the next examples will show.

Example 4.1

- If we take as seed $Z_0 = 1234$, then we will get the sequence of numbers
  \[ 0.1234, 0.5227, 0.3215, 0.3362, 0.3030, 0.1809, 0.2724, 0.4201, 0.6484, 0.0422, 0.1780, 0.1684, 0.8358, 0.8561, 0.2907, \ldots \]

- If we take as seed $Z_0 = 2345$, then we get the sequence of numbers
  \[ 0.2345, 0.4990, 0.9001, 0.0180, 0.0324, 0.1049, 0.1004, 0.0080, 0.0064, 0.0040, \ldots \] Two successive zeros behind the decimal point will never disappear.

- If we choose $Z_0 = 2100$, then we get the sequence
  \[ 0.2100, 0.4100, 0.8100, 0.6100, 0.2100, 0.4100, \ldots \] Only after four numbers the sequence starts to repeat itself.

4.1.2 Linear Congruential Method

Nowadays, most of the random number generators in use are so-called linear congruential generators. They produce a sequence of integers between 0 and $m-1$ according to the following recursion:

$$ Z_i = (aZ_{i-1} + c) \mod m, \quad i = 1, 2, 3, \ldots $$

The initial value $Z_0$ is called the seed, $a$ is called the multiplier, $c$ the increment and $m$ the modulus. To obtain the desired uniformly (0,1) distributed random numbers we should choose $U_i = Z_i / m$. Remark that $U_i$ can be equal to zero, which will cause some problems if we, for example, are going to generate exponentially distributed random variables (see section 4). A good choice of the values $a$, $c$ and $m$ is very important. One can prove that a linear congruential generator has a maximal cycle length $m$ if the parameters $a$, $c$ and $m$ satisfy the following properties:

- $c$ and $m$ are relatively prime;
- if $q$ is a prime divisor of $m$ then $a = 1 \mod q$;
- if 4 is a divisor of $m$ then $a = 1 \mod 4$.

Example 4.2

- If we choose $(a, c, m) = (1, 5, 13)$ and take as seed $Z_0 = 1$, then we get the sequence of numbers $1, 6, 11, 3, 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, \ldots$ which has maximal cycle length of 13.

- If we choose $(a, c, m) = (2, 5, 13)$ and take as seed $Z_0 = 1$, then we get the sequence of numbers $1, 7, 6, 4, 0, 5, 2, 9, 10, 12, 3, 11, 1, \ldots$ which has cycle length of 12. If we choose $Z_0 = 8$, we get the sequence $8, 8, 8, \ldots$ (cycle length 1!)