Notes on Stochastic Simulation

A. Ridder

June 6, 2022

Abstract

These notes give a short introduction in discrete-event simulation in stochastic operations research, and specifically, in queueing models. This includes how to estimate steady-state performance measures. Furthermore, these notes summarize the basic principles of Monte Carlo simulation, and give some details on uniform and nonuniform random number generation.

Contents

1 Introduction

In stochastic operations research you consider mathematical models of real world dynamic systems that evolve randomly in time. For instance queueing models are used to analyse call centers, or the Internet. We assume that you have some basic background of these mathematical models. The purpose of the analysis is usually to obtain certain performance measures of the system acting as indicators of the perceived 'quality of service'. Performance measures are typically

- expected average waiting time;
- expected average sojourn time;
- expected average waiting line;
- throughput;
- utilization;
- probability of a lost customer;
- probabilty of a customer who has to wait longer than xx time;

- etc.

To obtain these performance measures in complex systems, one often resorts to simulation of the associated mathematical model. Below we describe the discrete-event simulation method which is typically used for this purpose.

2 Monte Carlo Algorithm

The basics of stochastic simulation is an experiment of a system that involves a random input X and produces a (random) output $Y = h(X)$ as response to the input. The objective is to estimate certain performance measures such as $\mathbb{E}[Y]$, $Var(Y)$, or distributional measures ($\mathbb{P}(Y \in \cdot)$). Estimates of thes performance measures are obtained by averaging independent and identically distributed (IID) replications $Y_1, Y_2, \ldots$ of the output. This is based on the strong law of large numbers (SLLN):

Theorem 1. *(SLLN) When $Y_1, Y_2, \ldots$ are IID replications of Y then*

$$\frac{1}{n}\sum_{i=1}^{n} Y_i \overset{\text{a.s.}}{\to} \mathbb{E}[Y] \quad (n \to \infty).$$

Consider finitely many IID replications $Y_1, Y_2, \ldots, Y_n$. This is called a finite sample. Their average forms an approximation of the limit, thus we call it the sample average estimator, denoted

$$\overline{Y}_n \doteq \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

When n is large, then the SLLN says that $\overline{Y}_n \approx \mathbb{E}[Y]$. Note that $\overline{Y}_n$ is a random variable, thus any sample of size n would give another value, but these vary not much. Moreover, $\overline{Y}_n$ is an unbiased estimator, meaning $\mathbb{E}\left[\overline{Y}_n\right] = \mathbb{E}[Y]$. This leads to the Monte Carlo algorithm.

Algorithm 1 Monte Carlo Simulation

1: **for** $i = 0$ to n **do**
2: Generate an input X_i, independently of previous runs.
3: Compute output $Y_i = h(X_i)$.
4: **end for**
5: **return** Sample average $\overline{Y}_n$

The input X and the response function h might be rather involved in stochastic systems. This will be the topic of the next sections. In Section 10 it will be explained how to analyze the estimator by computing its variance, or even its distribution.

3 Discrete Event Simulation (DES)

Discrete-event simulation is a computer simulation model of a stochastic operational system that evolves dynamically in time. The system is described by states that change at discrete time epochs due to the occurrence of events. The key concepts are (i) time variable; (ii) system state; (iii) events and event list; (iv) state transitions; (v) clock structure; (vi) data registers. The first five concepts define the DES as a GSMP (general semi-Markov process), and enables simulating sample paths. The data registers are needed for computing the performance measures from the simulated sample paths.

4 The Standard Procedure

You simulate by moving forward the time variable from event time to the next event time, and by updating state and event list at these event times. In this way you simulate the model from time zero until some predefined stopping time, for instance, a finite time horizon T, a number of arrived or served customers K, a number of occurred events N, etc. This is called a simulation run which ends with collecting statistics associated with the performance measures, such as the total (or cumulative) waiting time, the queueing area, the number of customers that went into service, the number of customers that abandonned the queue before being served, etc. For instance, the number of served customers K and their total waiting time W yields the average waiting time per customer W/K in a run. For obtaining a reliable estimate of the mean waiting time you simulate many runs and take the average of all these W/K.

5 An Illustrative Example

As a reference model we consider a queueing network model, pictured in Figure 1.

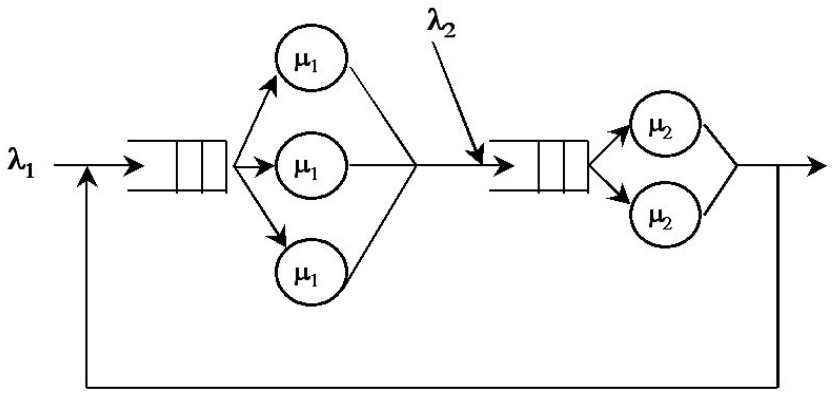


Figure 1: The reference queueing network.

- The queueing model consists of two stations ('queues') Q_1 and Q_2.
- At Q_i jobs arrive according to a Poisson(λ_i) process.
- There are three servers at Q_1 and two servers at Q_2.
- The service times of the servers at Q_i have distribution function $G_i(\cdot)$.
- After service completion at Q_1 the jobs enter Q_2.
- After service completion at Q_2 a job leaves the system with probability p or re-enters ('feeds back' to) Q_1 with probability $1-p$.
- Both stations have infinite waiting spaces.
- Waiting jobs are served in order of arrival (FCFS).

Typically we are interested in

- waiting times or system times at the two stations;

- waiting lines or system lines at two stations;
- time spend in the system (sojourn time);
- throughput (per station or from the whole system);
- utilization;
- ...

Furthermore, we have to specify

- Whether we wish to estimate these performance measures for a finite period; for instance the queueing system operates daily, opening empty at 8.00 hr in the morning, closing at 18.00 hr in the evening. We call this transient performance measures, and transient simulation.
- Or whether we wish to estimate steady-state averages; then we assume that the system operates for an infinite time. We call this staedy-state performance measures, and steady-state simulation.

5.1 DES Ingredients

5.1.1 System State

The system state is a collection of variables necessary to describe a system at a particular time. The set of all states is denoted by $\mathcal{X}$; a specific state by $\boldsymbol{x} \in \mathcal{X}$; and the (random) state at time t by $\boldsymbol{X}(t) \in \mathcal{X}$. In our example the state comprises

- the number of jobs (x_1, x_2) present at the two stations;
- two vectors a_1, a_2 of the arrival times of these waiting jobs;
- two vectors b_1, b_2 specifying the status of the servers (idle/busy).

5.1.2 Events

An event is an instantaneous occurrence that may change the state or trigger a state transition. The set of all possible events is denoted by $\mathcal{E}$; a specific event by e; the events active in state $\boldsymbol{x} \in \mathcal{X}$ by $E(\boldsymbol{x}) \subset \mathcal{E}$. In our example events are

- the arrival of a new job (at Q_1 or at Q_2);
- a service completion at one of the five servers.

5.1.3 Time or Clock Variable

Events occur at some point in time. For this we need a variable representing the current time of the simulation. This is also called the simulation clock or system time that measures the elapsed simulation time. Clearly we need to specify its unit size (second or minute or ...); then we set it to zero at the start of a simulation and update it every time an event occurs.

5.1.4 Event List or Calendar

The calendar for the simulation is a list of the events that are currently scheduled to occur. There is only one event list and it consists of the scheduled event times, sorted by the earliest scheduled time first. The event list at time t is denoted by $L_{\text{ev}}(t)$. In our example the event list comprises

- the next arrival time of a customer at Q_1;
- the next arrival time of a customer at Q_2;
- for any busy server the next time he is ready completing the job.

Another view of the event list is that it is a collection of alarm clocks, one for each scheduled event. The clocks are preset at different (random) times in the future. For instance at some t there are 3 events scheduled, as shown in Figure 2.

$L_{\text{ev}}(t) = \Big\{$    $\Big\}$

arrival Q_1 arrival Q_2 departure Q_2

Figure 2: Eventlist represented by alarm clocks.

5.2 The Event-Scheduling Approach

The discrete-event simulation runs as follows.

- The simulation clock is advanced forward to the earliest time on the event list (when the first alarm goes off).
- The alarm belongs to a particular event that triggers several activities in the system.
- These activities make that there will be changes in the system state and in the event list; these need to be updated.
- Then the simulation clock is advanced again, etc.

5.3 A Walk Through DES

In our queueing example we let time be measured in seconds, starting at 08:00. This is system time 0.

- Suppose current time is t_{sim} and state $\boldsymbol{x} \in \mathcal{X}$.
- Each active event $e_i \in E(\boldsymbol{x})$ has an associated scheduled alarm time t_i.
- We denote the type-time pair of events by ['A1', `<time>`] for the arrival event at time `<time>` at Q_1 (and similarly for arrival event at Q_2);
- and by ['D1', 1, `<time>`] for departure events due to service completion at server 1 of Q_1; similarly for the other servers, and the other queue.

- We start $t_{\text{sim}} = 0$ with an empty system and idle servers:

$$\boldsymbol{x} = \{x_1 = 0, x_2 = 0, a_1 = [], a_2 = [], b_1 = (0, 0, 0), b_2 = (0, 0)\}.$$

- There are two active events: arrivals at the queues; the associated event times are drawn by our random number generator, resulting. Thus

$$L_{\text{ev}} = \{[\text{'A2'}, 0.817], [\text{'A1'}, 1.284]\}.$$

5.3.1 The 1-st Event

- The simulation time is advanced to the earliest event time 0.817 belonging to the event of an arriving job at Q_2;
- Immediately, this job enters service with server 1 of Q_2;
- We draw a sample of the service time S_2, say 1.693, to schedule a new event 'D2' with departure time $0.817 + 1.693 = 2.510$;
- Also we realise a new sample of the interarrival time A_2, say 1.226, and update the scheduled time of event 'A2' to $0.817 + 1.226 = 2.043$;

Thus the new situation is:

$$\begin{aligned} t_{\text{sim}} &= 0.817 \\ \boldsymbol{x} &= \{x_1 = 0, x_2 = 1, a_1 = [], a_2 = [], b_1 = (0, 0, 0), b_2 = (1, 0)\} \\ L_{\text{ev}} &= \{[\text{'A1'}, 1.284], [\text{'A2'}, 2.043], [\text{'D2'}, 1, 2.510]\} \end{aligned}$$

5.3.2 The 2-nd Event

- The simulation time is advanced to the earliest event time 1.284 belonging to the event of an arriving job at Q_1.
- This job enters service with server 1 of Q_1 .
- We draw a sample of the service time S_1, say 2.613, to schedule a new event 'D1' with departure time $1.284 + 2.613 = 3.987$;
- We realise a new sample of the interarrival time A_1, say 0.577, and update the scheduled time of event 'A1' to $1.284 + 0.577 = 1.861$;

Thus the new situation is:

$$\begin{aligned} t_{\text{sim}} &= 1.284 \\ \boldsymbol{x} &= \{x_1 = 1, x_2 = 1, a_1 = [], a_2 = [], b_1 = (1, 0, 0), b_2 = (1, 0)\} \\ L_{\text{ev}} &= \{[\text{'A1'}, 1.861], [\text{'A2'}, 2.043], [\text{'D2'}, 1, 2.510], [\text{'D1'}, 1, 3.987]\} \end{aligned}$$

5.3.3 The 3-rd Event

- The simulation time is advanced to the earliest event time 1.861 belonging to the event of an arriving job at Q_1.
- This job enters service with server 2 of Q_1.

- We draw a sample of the service time S_1, say 0.811, to schedule a new event 'D1' with departure time $1.861 + 0.811 = 2.752$;
- We realise a new sample of the interarrival time A_1, say 1.428, and update the scheduled time of event 'A1' to $1.861 + 1.428 = 3.289$;

Thus the new situation is:

$$\begin{aligned} t_{\text{sim}} &= 1.861 \\ \boldsymbol{x} &= \{x_1 = 2, x_2 = 1, a_1 = [], a_2 = [], b_1 = (1, 1, 0), b_2 = (1, 0)\} \\ L_{\text{ev}} &= \{[\text{'A2'}, 2.043], [\text{'D2'}, 1, 2.510], [\text{'D1'}, 2, 2.752], [\text{'A1'}, 3.289], [\text{'D1'}, 1, 3.987]\} \end{aligned}$$

5.3.4 The k-th Event

Suppose that currently the situation is

$$\begin{aligned} t_{\text{sim}} &= 249.31 \\ \boldsymbol{x} &= \{x_1 = 4, x_2 = 8, a_1 = (240.82), a_2 = (238.71, 240.61, 241.01, 244.55, 246.91, 248.88), \\ &\quad b_1 = (1, 1, 1), b_2 = (1, 1)\} \\ L_{\text{ev}} &= \{[\text{'D2'}, 2, 250.28], [\text{'D2'}, 1, 251.43], [\text{'A1'}, 254.38], [\text{'D1'}, 2, 255.36], \\ &\quad [\text{'A2'}, 256.42], [\text{'D1'}, 1, 260.91], [\text{'D1'}, 3, 263.93]\} \end{aligned}$$

- The next event is service completion at Q_2 at time 250.28.
- There are jobs waiting at Q_2 to occupy the empty seat. We draw a sample of the service time S_2, say 0.83, and update the scheduled time of event 'D2', 2 to $250.28+0.83 = 251.11$.
- For the job leaving Q_2 we flip a coin (Bernoulli random variable) to decide a feed back; suppose the outcome is to loop back for entering Q_1 again.
- The looped job finds all servers busy at Q_1, and thus joins the queue.

Next situation:

$$\begin{aligned} t_{\text{sim}} &= 250.28 \\ \boldsymbol{x} &= \{x_1 = 5, x_2 = 7, a_1 = (240.82, 250.28), a_2 = (240.61, 241.01, 244.55, 246.91, 248.88), \\ &\quad b_1 = (1, 1, 1), b_2 = (1, 1)\} \\ L_{\text{ev}} &= \{[\text{'D2'}, 2, 251.11], [\text{'D2'}, 1, 251.43], [\text{'A1'}, 254.38], [\text{'D1'}, 2, 255.36], \\ &\quad [\text{'A2'}, 256.42], [\text{'D1'}, 1, 260.91], [\text{'D1'}, 3, 263.93]\} \end{aligned}$$

5.3.5 Trace

This detailed walk through a system simulation is called a trace. In a trace a print is made of all the (important) simulation components after each event. It serves two objectives.

(i). In developing a simulation program it helps you to think about the events that may happen in the system, the activities they trigger and their logic.

(ii). After having written a computer program of the simulation, it provides a check whether the program is correct.

5.4 Simulation Run

A trace covers usually a short time interval or a small number of events. The actual execution of the computer program of the simulation is longer but should end at some time or after some number of events: stopping criterion. One such a simulation is called a simulation run which corresponds to the concept of sample path of a stochastic process. Typically, you will simulate (many) more simulation runs in order to obtain reliable estimates.

5.4.1 Programming a Run

A simulation run of the system goes from event to event at discrete time epochs determined by realisations of the appropriate random variables. When you write a computer program of the simulation it is convenient to modularise your program into several subprograms or routines to clarify the logic and the interactions. Roughly a simulation run looks in pseudo-code as follows.

```
initialise;
REPEAT
    [e,t] = next_event_type_and_time;
    switch (e)
        case 'arrival': ...
        case 'departure': ...
        case '...': ...
UNTIL stopping_criterion;
```

In the subroutines for the different events you program code for updating state, event list, and statistical counters. Similar as what you would do in the trace. For instance:

```
subroutine execute_arrival_at_Q1_event(t)
begin
    update_all_relevant_counter_variables();
    j = find_free_server();
    if (server(j)=='idle')
        server(j) = 'busy';
        s = generate_servicetime();
        add_to_eventlist('D1',j,t+s);
    else
        add_customer_to_queue(j,t);
    end
    a = generate_interarrivaltime();
    add_to_eventlist('A1',t+a);
end
```

When does a run ends? For now, we consider transient simulation; i.e, there is a natural finite horizon where the run ends. Alternatively, the stopping criterion might be that a certain number of events has reached, or a certain number of arriving customers, or, etc.

5.4.2 Statistical (or Counter) Variables in a Run

In each run you keep track of certain variables, called statistical variables, or counter variables. At the end of the run these variables determine the outcomes or outputs of the run. Which variables you need depend on the intended performance measures.

In finite-horizon simulation of a queueing system typical performance measures are

- The expected average waiting time per customer.
- The expected average queue length per unit of time.
- The througput (this is the expected average number of service completions per unit of time).
- Utilization (this is the expected fraction of time that the servers are busy).

The counter variables are initialised at the start of a simulation run, and because nothing changes in between two consecutive events, it suffices to update them after an event occurs.

Example 1. For instance, consider Q_1 in the tandem network. Suppose that the system runs during 10 hours, and that the objective is to estimate the performance measure "expected average waiting time per customer" (during these ten hours). This means:

- Let K be the (random) number of customers that has been served during these ten hours at Q_1.
- Let $W_1, \ldots, W_K$ be their corresponding (random) waiting times (possibly some are zero).
- Define the average waiting time per customer as the output $Y = (1/K) \sum_{j=1}^{K} W_j$.
- then we wish to compute $\mathbb{E}[Y]$.

A simulation run of the queueing network should result in an observation y of the random object Y. Thus, during a run in the simulation program you keep track of two variables:

1. n_{serv1} = the current total number customers who went into service at Q_1.
2. w_{total1} = the current total waiting time of these n_{serv1} customers.

Whenever a new customer enters service at Q_1, these two variables are updated. For this, you need to compute the waiting time of the customer, which is possible when you include arrival times of waiting customers in the system state. At the end of the run you have a realisation $y = w_{\text{total1}}/n_{\text{serv1}}$ of the output Y.

Example 2. Now suppose that the objective is to estimate the performance measure "expected average queue length at station Q_1 per unit of time" (during these ten hours). This means:

- Let $Q_1(t)$ be the queue length at station 1 at time t, $0 \leq t \leq T$ (time is measured in some unit, e.g. minutes, or hours; T is such that it represents 10 hours).
- Define the average queue length per unit of time as output $Y = \frac{1}{T} \int_0^T Q_1(t)\, dt$.
- Goal: $\mathbb{E}[Y]$.

A simulation run of the queueing network should result in an observation y of the random object Y. To do this, rewrite Y to reflect that there is no change in the queue size in between two consecutive event times, see Figure 3.

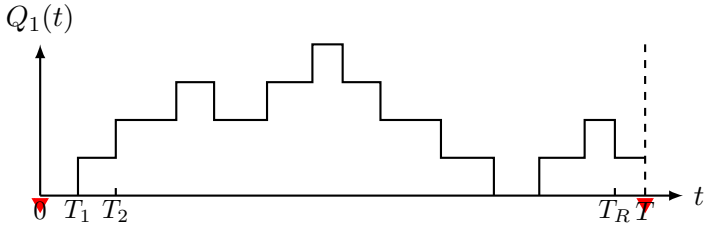


Figure 3: The queue size at station Q_1 on $[0, T]$.

Let $0 = T_0 < T_1 < T_2 < \cdots < T_K < T \leq T_{K+1}$ be the consecutive event times, where the total number of events K in $[0, T]$ is random. Then, the average queue length per unit time is

$$Y = \frac{1}{T} \sum_{k=1}^{K} (T_k - T_{k-1}) Q(T_{k-1}) + \frac{1}{T} (T - T_K) Q(T_K).$$

Thus, during a run in the simulation program you keep track of one variable:

A: total area under the graph upto the current event time.

Whenever an event occurs this variable is updated. For this, you need to compute the duration between the previous and current event times, which follows simply by the clock variable; and you need the queue length, which is possible when you include it in the system state. At the end of the run you have a realisation $y = A/T$ of the output Y.

5.5 Many Runs

To get estimates of the performance measures, you execute n runs, independently and (probabilistic) identically. The n runs produce IID outputs $Y_1, \ldots, Y_n$, see Figure4. The performance measures are estimated by averaging these n outcomes. The statistics of this averaging are given in Section 10.

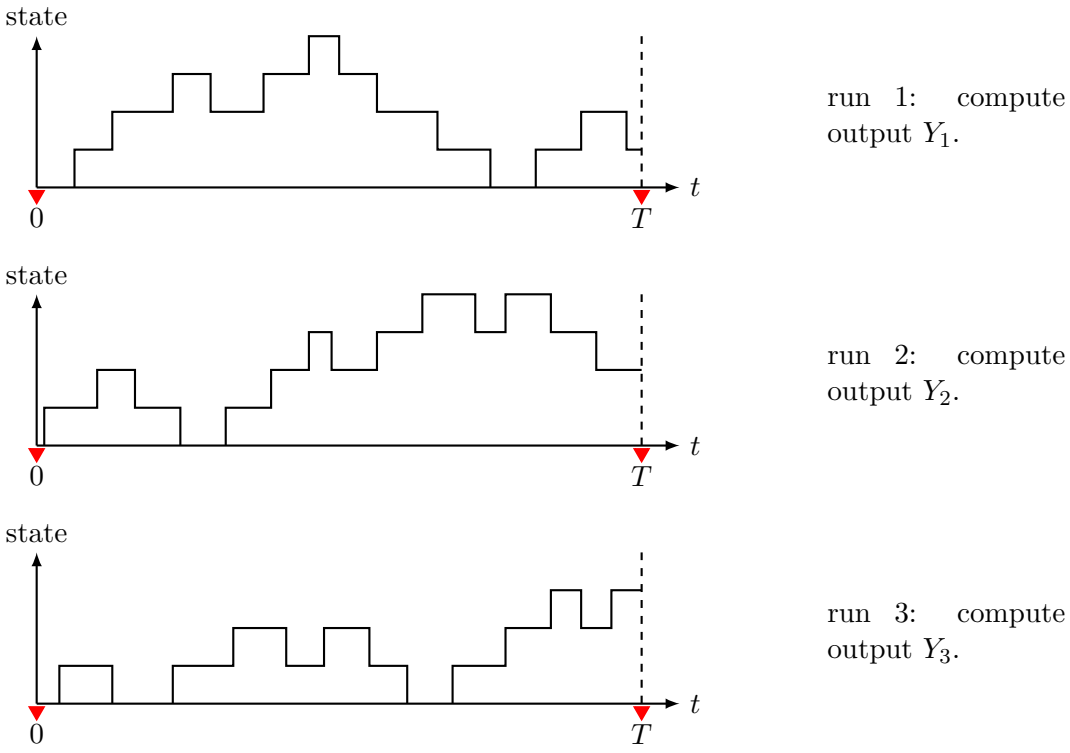


Figure 4: Illustration of 3 simulation runs.

6 $G/G/1$ Queue

The single server queue is easy to implement in a simulation program. The event list consists of at most two numbers only, viz., the next scheduled arrival time and the next scheduled service completion time (or departure time from the system). In a conputer program you might

implement the eventlist as an array of fixed length with two components (arrival and depature events). Whenever the server is idle, the departure event is set to infinity. Alternatively, the eventlist is a dynamic array, adapted to the scheduled events only. The state contains usually the number of customers present (or the number in the queue), the status of the server (busy or idle) and the arrival times of customers in the queue. The latter is implemented as a dynamic array.

7 $G/G/c$ Queue

There are two types of events, arrivals and service completions (or departures). Since there is a single arrival process, there is a single arrival event. But there may be up to c scheduled departure events dependent on how many servers are busy. The data structure of the event list is a linked list of a dynamic number of nodes. Each node represents a scheduled event and should contain information on the scheduled event time, i.e., the time the event is due to occur, the event type, i.e., arrival or departure, and possibly other information (for instance a reference to the specific server from which the departure is going to be). Notice that the event list always contains at least one node associated with the arrival event.

The data structure of the queue of waiting customers is a linked list of a dynamic number of nodes. Each node represents a a customer waiting in line and should contain information on his arrival time, i.e., the time he joined the queue, and possibly other information (for instance, the required service time, or its priority, etc).

Let $Q(t)$ be the number of customers in the queue at time t in the stochastic model of the system, W_k the waiting time of the k-th customer (possibly zero!), and K the number of customers whose service started in $[0, T]$. Then typically, the performance measures that are estimated are

$$\text{expected average waiting time} = \mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K} W_k\right]$$

$$\text{expected average queue size} = \mathbb{E}\left[\frac{1}{T}\int_0^T Q(t)\,dt\right]$$

$$\text{expected fraction of delayed customers} = \mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K} 1\{W_k > 0\}\right]$$

$$\text{expected fraction of customers served on time} = \mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K} 1\{W_k \le \tau\}\right]$$

8 A Python Code of a Queue Simulation

Here is a program of a transient simulation of the $M/M/c$ queue. The program is for estimating transient performance measures average waiting time, average queue length, delay probabilty (fraction of customers who has to wait), and the service level (fraction of customers who are served within some specified time, for instance half a minute). Transient means that is defined by a given finite time period, or a given finite number of events, or a given finite number of customers, or etc. Below we consider a given finite duration T.

In the code:

- System state consists of two structures; (i) $X = (x_1, x_2)$, where x_1 = number of waiting customers in queue, and x_2 = number of busy serve rs; (ii) $Q = (q_1, \ldots)$ is the queue of waiting customers; q_i contains information of the i-th waiting customer, namely arrival time, required service time, and an identification number. The queue is ordered according to arrival times. $Q = ()$ if no one is waiting.
- Events are arrivals and departures (service completions).
- Event list is $L = (e_1, \ldots)$, where e_j is the j-th scheduled event containing information of time to occur, type, and an identity number. The list is ordered according to the scheduled times.
- Collected counter variables during a run is a vector of observations $V = (v_1, \ldots, v_6)$ for collecting queueing area, waiting times, number arrivaed, number of served, number to wait, number on time.
- Customers, events, queue, eventlist, and observation vector are implemented as `class`.

```
1  ########## packages ######################
2  import numpy as np
3  import numpy.random as rnd
4
5
6  ########## classes ######################
7  # customer has identification number, and
8  # arrival time, and service time as a 3-tuple info
9  class Customer(object):
10     def __init__(self, _idnr, _arrtime = np.inf,
11                        _sertime = np.inf):
12         self.info = (_idnr, _arrtime, _sertime)
13
14     def getIdnr(self):
15         return self.info[0]
16
17     def getArrTime(self):
18         return self.info[1]
19
20     def getSerTime(self):
21         return self.info[2]
22
23 # list of waiting customers
24 class Queue(object):
25     def __init__(self, _line = []):
26         self.line = _line
27
28     # adds at the end
29     def addCustomer(self, cust):
30         if len(self.line)==0:
31             self.line = [cust]
32         else:
33             self.line.append(cust)
```

```
34
35      # returns and deletes customer with idnr i
36      def deleteCustomer(self,i):
37          cust = next(cus for cus in self.line if cus.getIdnr()==i)
38          self.line.remove(cust)
39          return cust
40
41      # gets the customer in front and removes from queue
42      def getFirstCustomer(self):
43          cust = self.line.pop(0)
44          return cust
45
46  # events are arrivals, and departures from agents
47  # as 3-tuple info (time of event, type of event, and idnr
48  # idnr for arrival is 0
49  # idnr for departures is customer idnr
50  class Event(object):
51      def __init__(self, _time = np.inf, _type = '', _idnr = 0):
52          self.info = (_time, _type, _idnr)
53
54      def getTime(self):
55          return self.info[0]
56
57      def getType(self):
58          return self.info[1]
59
60      def getIdnr(self):
61          return self.info[2]
62
63  # events are ordered chronologically
64  # always an arrival event
65  class Eventlist(object):
66      def __init__(self, _elist = []):
67          self.elist = _elist
68
69      # adds according to event time
70      def addEvent(self, evt):
71          if len(self.elist)==0:
72              self.elist = [evt]
73          else:
74              te = evt.getTime()
75              if te > self.elist[-1].getTime():
76                  self.elist.append(evt)
77              else:
78                  evstar = next(ev for ev in self.elist
79                                if ev.getTime()>te)
80                  evid = self.elist.index(evstar)
81                  self.elist.insert(evid, evt)
82
```

```
83      # returns oldest event and removes from list
84      def getFirstEvent(self):
85          evt = self.elist.pop(0)
86          return evt
87
88      # deletes event associated with customer with idnr i
89      def deleteEvent(self,i):
90          evt = next(ev for ev in self.elist if ev.getIdnr()==i)
91          self.elist.remove(evt)
92
93  # counter variables per run
94  # queueArea = int Q(t)dt where Q(t)=length of queue at time t
95  # waitingTime = sum W_k where W_k= waiting time of k-th customer
96  # numArr = number of arrivals
97  # numServed = number served
98  # numOntime = number of customers startng their service on time
99  # numWait = number of arrivals who go into queue upon arrival
100 class Qobservations(object):
101     def __init__(self, _queueArea = 0, _waitingTime = 0,
102     _numArr = 0, _numServed = 0, _numOntime = 0, _numWait = 0):
103         self.queueArea  = _queueArea
104         self.waitingTime = _waitingTime
105         self.numArr = _numArr
106         self.numServed = _numServed
107         self.numOntime = _numOntime
108         self.numWait = _numWait
109
110     def addObs(self,obs):
111         self.queueArea  += obs.queueArea
112         self.waitingTime += obs.waitingTime
113         self.numArr += obs.numArr
114         self.numServed += obs.numServed
115         self.numOntime += obs.numOntime
116         self.numWait += obs.numWait
117
118 # state is queue length and number of busy servers
119 class State(object):
120     def __init__(self, _queueLength = 0, _numBusy = 0):
121         self.queueLength = _queueLength
122         self.numBusy = _numBusy
123
124 ############### random stuff ############################
125 # exponential variate
126 # using the random number generator rng (defined in main)
127 def ranexp(rate,rng):
128     return -np.log(rng.random()) / rate
129
130 ############## arrival and departure routines #########
131 # lamda = arrival rate
```

```
# mu = service rate
# c = number of servers
# te = time of arrival event
# nc = customer number
# X = state (queue length and busy servers)
# Q = state (queue of customers)
# L = eventlist
# obs = vector of counter variables

def HandleArrival(lamda,mu,c,te,nc,X,Q,L,obs,rng):
    nc += 1
    ser = ranexp(mu,rng) # required service time

    eva = Event(te + ranexp(lamda,rng),'arr',0) # next arrival
    L.addEvent(eva)
    if X.numBusy < c: # there is a free agent
        X.numBusy += 1
        obs.numServed += 1
        obs.numOntime += 1
        evb = Event(te + ser, 'dep', nc) # departure from agent
        L.addEvent(evb)
    else: # all agents busy, customer joins queue
        cust = Customer(nc,te,ser)
        Q.addCustomer(cust)
        X.queueLength += 1
        obs.numWait += 1

    return nc

def HandleDeparture(AWT,te,X,Q,L,obs):
    if X.queueLength == 0: # no one is waiting
        X.numBusy -= 1
    else: # takes the first in line
        first = Q.getFirstCustomer()  # delete from queue
        X.queueLength -= 1
        w = te - first.getArrTime() # waiting time
        obs.waitingTime += w
        if w < AWT:
            obs.numOntime += 1
        obs.numServed += 1
        i = first.getIdnr()
        ser = first.getSerTime()
        evb = Event(te + ser, 'dep', i) # departure from agent
        L.addEvent(evb)

########## simulations ####################
# simulation run until end of day T
# all random times are exponential
# AWT = accepted waiting time
```

```
181
182 def simrun(lamda,mu,c,T,AWT,rng):
183     tc = 0 # current time
184     nc = 0 # customer number = number of arrivals
185
186     X = State()
187     Q = Queue()
188     L = Eventlist()
189     obs = Qobservations()
190
191     evt = Event(ranexp(lamda,rng),'arr',0) # first arrival
192     L.addEvent(evt)
193
194     while tc < T:
195         evt = L.getFirstEvent() # next event
196         te = evt.getTime()
197         tp = evt.getType()
198         ti = te - tc
199         obs.queueArea  += ti * X.queueLength
200
201         if tp == 'arr': # arrival event
202             nc = HandleArrival(lamda,mu,c,te,nc,X,Q,L,obs,rng)
203         else: # departure event
204             HandleDeparture(AWT,te,X,Q,L,obs)
205         tc = te
206
207     obs.queueArea /= te # average queue length
208     obs.waitingTime /= obs.numServed # average waiting time
209     obs.numArr = nc
210     obs.numOntime /= obs.numServed # fraction served on time
211     obs.numWait /= nc # prob of waiting (delay prob)
212
213     return obs
214
215 # do n runs, collect statistics and report output
216 def simulations(lamda,mu,N,T,AWT,n,rng):
217     cumobs = Qobservations()
218     for j in range(n):
219         obs = simrun(lamda,mu,N,T,AWT,rng)
220         cumobs.addObs(obs)
221
222     L = cumobs.queueArea/n
223     W = cumobs.waitingTime/n
224     PW = cumobs.numWait/n
225     SL = cumobs.numOntime/n
226
227     print('average queue length L :', L)
228     print('average waiting time W :', W)
229     print('delay probability P_w  :', PW)
```

```
230     print('service level SL       :', SL)
231
232 ########## main ####################
233 def main():
234     mu = 0.2 # per minute: 5 minutes average service
235     rho = 0.8 # utilization
236     c = 5 # number of servers
237     lamda = rho * c * mu # arrivals per minute
238     print('arrival rate lamda =', lamda)
239     AWT = 0.5 # half minute
240     T = 300.0 # minutes (5 hrs)
241     n = 200 # number of runs
242
243     seed = 12345 # choose your own or none
244     rng = rnd.default_rng(seed)
245
246     simulations(lamda,mu,c,T,AWT,n,rng)
247
248 if __name__ == '__main__':
249     main()
```

Output

```
arrival rate lamda = 0.8
average queue length L : 1.9771138629463751
average waiting time W : 2.3947285520297474
delay probability P_w  : 0.5338730049286071
service level SL       : 0.5229032594308
```

Note that these are just point esitmates. Better reporting includes standard errors, confidence intervals, etc. The identity numbers given to customers and events are not really used in this program. But these can be useful when you need to identify customers or events. For instance in case of abandonments.

9 Poisson-Exponential Queueing Models

In case all scheduled event times have exponential distributions, it is possible to exploit three properties of the exponential distribution. Check your Probability textbook.

Lemma 1. *Let X be a random probability, exponentially distributed, then*

$$\mathbb{P}(X > s + t | X > s) = \mathbb{P}(X > t) \ \ (\textit{for all } s, t > 0).$$

This is the so-called memoryless property.

Lemma 2. *Let $X_1, X_2, \ldots, X_n$ be n independent random variables, exponentially distributed with rate λ_i (for X_i), $i = 1, \ldots, n$. Define*

$$\tau \doteq \min\{X_1, \ldots, X_n\}.$$

Then τ is exponentially distributed with rate $\lambda_1 + \cdots + \lambda_n$.

Lemma 3. *Let $X_1, X_2, \ldots, X_n$ be n independent random variables, exponentially distributed with rate λ_i (for X_i), $i = 1, \ldots, n$. Define*

$$\tau \doteq \min\{X_1, \ldots, X_n\}.$$

Then

$$\mathbb{P}(\tau = X_i) = \frac{\lambda_i}{\lambda_1 + \cdots + \lambda_n} \quad (i = 1, \ldots, n).$$

Thus, suppose that at some event time T_k, certain events $e_1, \ldots, e_n$ are scheduled to occur, where event e_i is associated with an exponentially distributed time with rate λ_i $(i = 1, \ldots, n)$.

- Then, you generate a duration τ from the exponential distribution with rate $\lambda_1 + \cdots + \lambda_n$, and set the next event time $T_{k+1} = T_k + \tau$.
- And you let event e_i occur with probability $p_i = \lambda_i/(\lambda_1 + \cdots + \lambda_n)$, $i = 1, \ldots, n$.

This means that in the computer code of DES, you have the same system state and counter variables as previous, and you have to keep track of which events are scheduled, but you do not need to set the event times in the event list. This will speed up considerably the execution time of a simulation run when there are many events scheduled.

10 The Statistics of Transient Simulation

The purpose of simulation is to estimate a performance measure $\mathbb{E}[Y]$ for some random variable Y, for instance Y is the average waiting time per customer during an arbitrary working day (12 hours) at a call center. Denote $\mu = \mathbb{E}[Y]$ and $\sigma^2 = \mathrm{Var}[Y]$, both unknown. The underlying theories that you apply are the law of lare numbers and the central limit theorem. Suppose that $Y_1, Y_2, \ldots$ are i.i.d. random variables with the same distribution as Y, and define for $n \in \mathbb{N}$

$$\widehat{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i, \quad S_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(Y_i - \widehat{Y}_n)^2.$$

These are called sample average and sample variance, based on sample size n. Then

$$\mathbb{E}\left[\widehat{Y}_n\right] = \mu \tag{1}$$

$$\mathrm{Var}\left[\widehat{Y}_n\right] = \frac{1}{n}\sigma^2 \tag{2}$$

$$\mathbb{P}\left(\lim_{n\to\infty} \widehat{Y}_n = \mu\right) = 1 \tag{3}$$

$$\mathbb{E}\left[S_n^2\right] = \sigma^2 \tag{4}$$

$$\lim_{n\to\infty} \mathbb{P}\left(|S_n^2 - \sigma^2| > \epsilon\right) = 0 \tag{5}$$

$$\lim_{n\to\infty} \mathbb{P}\left(\frac{\widehat{Y}_n - \mu}{\sigma/\sqrt{n}} \le x\right) = \Phi(x), \tag{6}$$

$$\frac{\widehat{Y}_n - \mu}{\sigma/\sqrt{n}} \stackrel{\mathrm{D}}{=} N(0,1) \;\Rightarrow\; \mathbb{P}\left(\frac{\widehat{Y}_n - \mu}{S_n/\sqrt{n}} \le x\right) = T_{n-1}(x), \tag{7}$$

$$\lim_{n\to\infty} \mathbb{P}\left(\frac{\widehat{Y}_n - \mu}{S_n/\sqrt{n}} \le x\right) = \Phi(x), \tag{8}$$

where $\Phi(\cdot)$ is the standard normal cumulative probability distribution function, and $T_{n-1}(\cdot)$ is the standard Student's-t probability distribution function with $n-1$ degrees of freedom. Properties (1)-(8) have the following interpretations.

- Equation (1) says that the sample average $\widehat{Y}_n$ is an unbiased estimator of μ (for any n).
- Equation (3) is the law of large numbers which says that almost all realisations of the sample average are close to the unknown μ (for large n).
- Equation (6) is the central limit theorem which says that the sample average is approximately normal distributed (for large n).
- Equation (4) says that the sample variance S_n^2 is an unbiased estimator of σ^2.
- Equation (5) says that the sample variance S_n^2 is a strongly consistent estimator of σ^2.
- Equation (8) follows from (5) and (6). It says that the statistic $\sqrt{n}(\widehat{Y}_n - \mu)/S_n$ has asymptotically a normal distribution.

When you have a simulation model that represents correctly the system, you make runs that are independent and identically distributed. A run is a simulation of the model during a single day (or some other time period which corresponds to the performance $\mathbb{E}[Y]$ of interest). The runs are different when the random number generator (RNG) is in different states at the starts of the runs (see Section 12). The i-th run gives you a realisation (or output, or sample, or value, or outcome) y_i of Y_i. These output variables $Y_1, Y_2, \ldots$ are i.i.d. and you may apply the properties (1)-(8). Specifically, let

$$\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i, \quad \bar{s}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2.$$

These are realisations of the sample average and the sample variance, respectively. Thus, $\bar{y}$ is an estimate of μ, and $\bar{s}^2$ is an estimate of σ^2. Notice that for large sample size n it is more efficient to calculate the sample variance by

$$\bar{s}^2 = \frac{1}{n-1}\sum_{i=1}^{n} y_i^2 - \frac{n}{n-1}\bar{y}^2.$$

Recall also that you use this (estimate of) the sample variance to estimate the standard error, i.e., the standard deviation $\sqrt{\mathrm{Var}[\widehat{Y}_n]}$ of the (sample average) estimator. Thus, the standard error is estimated by

$$\frac{\bar{s}}{\sqrt{n}} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} y_i^2 - \bar{y}^2} \Big/ \sqrt{n-1},$$

where $\frac{1}{n}\sum_{i=1}^{n} y_i^2$ is in fact an estimate of the second moment $\mathbb{E}[Y^2]$.

Furthermore, let $z_{1-\alpha/2}$ be the $(1-\alpha/2)$-quantile of the standard normal distribution, i.e., $\Phi(z_{1-\alpha/2}) = 1-\alpha/2$ (where $\alpha \in (0,1)$). Because $\Phi(-z_{1-\alpha/2}) = \alpha/2$, we get

$$\mathbb{P}\left(-z_{1-\alpha/2} < N(0,1) < z_{1-\alpha/2}\right) = \Phi(z_{1-\alpha/2}) - \Phi(-z_{1-\alpha/2}) = 1-\alpha.$$

A typical value of $\alpha = 0.05$ which gives $z_{1-\alpha/2} = z_{0.975} = 1.96$. Hence, from (8) we construct a $100(1-\alpha)\%$ confidence interval

$$\left(\bar{y} - z_{1-\alpha/2}\frac{\bar{s}}{\sqrt{n}}\,,\ \bar{y} + z_{1-\alpha/2}\frac{\bar{s}}{\sqrt{n}}\right).$$

For small sample sizes ($n < 250$) the normal approximation in (8) might be not valid in which case one usually uses the Student's-t critical values, i.e., let $t_{n-1,1-\alpha/2}$ be the $(1-\alpha/2)$-quantile of the Students-t distribution with $n-1$ degrees of freedom. Specifically,

$$\mathbb{P}\left(-t_{n-1,1-\alpha/2} < \text{Student's-}t < t_{n-1,1-\alpha/2}\right) = T_{n-1}(t_{n-1,1-\alpha/2}) - T_{n-1}(-t_{n-1,1-\alpha/2}) = 1-\alpha.$$

A typical value of $\alpha = 0.05$ which gives

n	10	50	100	250	1000
$t_{n-1,1-\alpha/2}$	2.262	2.010	1.984	1.970	1.962

Hence, we construct a $100(1-\alpha)\%$ confidence interval

$$\left(\bar{y} - t_{n-1,1-\alpha/2}\frac{\bar{s}}{\sqrt{n}}\,,\ \bar{y} + t_{n-1,1-\alpha/2}\frac{\bar{s}}{\sqrt{n}}\right).$$

Example 3. Three points are chosen randomly from the unit square. The expected area of the triangle that they form, is computed by simulation. The input of the system are three IID $X_1, X_2, X_3 \in [0,1]^2$. The output $Y = H(X_1, X_2, X_3)$ is the area of the triangle they form. There are several explicit formulas. We choose Heron's formula: Define

$$\begin{aligned} L_1 &= \text{distance between } X_1 \text{ and } X_2;\\ L_2 &= \text{distance between } X_1 \text{ and } X_3;\\ L_3 &= \text{distance between } X_2 \text{ and } X_3;\\ S &= \frac{1}{2}(L_1 + L_2 + L_3). \end{aligned}$$

Then

$$Y = \sqrt{S(S-L_1)(S-L_2)(S-L_3)}.$$

It is straightforward to program this formula. We get output (for sample size 1000):

```
sample size 1000
estimation 0.0781065
standard error 0.00210456
95% confidence interval ( 0.0739815 , 0.0822314 )
relative width 10.5623 %
```

To get smaller confidence interval, for instance about 5% relative width,

```
sample size 5000
estimation 0.0772075
95% confidence interval ( 0.0753025 , 0.0791125 )
relative width 4.93474 %
```

Do 200 experiments of size 1000 and check normality. Figure 5 shows the histogram of the 200 simulated data:

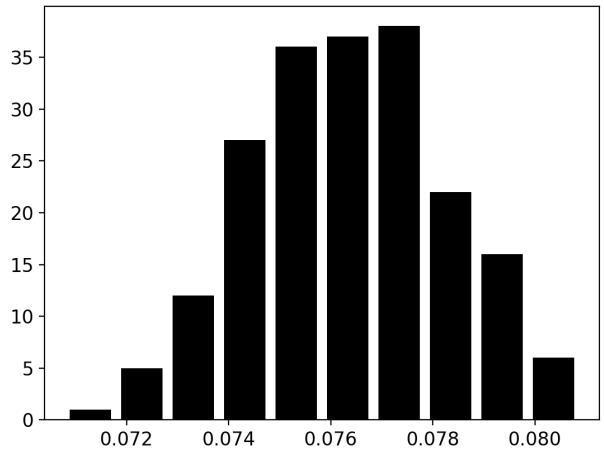


Figure 5: Histogam of 200 observations of the estimator.

The skewness and kurtosis of a (theoretical) normal distribution are 0. The data have

```
skewness=-0.033136810783961614
kurtosis=-0.234497625697625
```

The Jarque-Bera test for normality produces

```
test statistic 0.49484441144626456
pvalue 0.7808109610723597
```

These three diagnostics indicate indeed that the estimator based on avaraging 1000 runs, is (approximately) normal distributed.

11 Estimating Steady-State Performance Measures

Let $X(t)$ be the 'state' of the queueing system at time t. We keep it quite general and do not specify 'state'. Let H be the response function acting on the state; for instance

- $H(X(t)) =$ the number of customers present at time t;
- $H(X(t)) = 1$ if there is a waiting line at time t, and $H(X(t)) = 0$ if not;
- $H(X(t)) =$ the waiting time of the customer in front of the queue;
- $H(X(t)) = 1$ if the waiting time of the customer in front of the queue is less than some acceptable A_w, and $H(X(t)) = 0$ if it is larger;
- $H(X(t)) = 1$ if the server is busy at time t, and $H(X(t)) = 0$ else.

We call H the output function, and $H(X(t))$ the observation at time t. The assumption is that we can simulate the process $\{X(t) : 0 \leq t \leq T\}$ for a finite period of time, and that we can compute the observation $H(X(t))$ at any time instance. Assume that $X(t)$ converges (in distribution) to a limiting random variable $X = X(\infty)$ as $t \to \infty$, also called the steady-state variable. Denote

$$Y^{\text{st}} \doteq H(X) = \lim_{t\to\infty} H(X(t)).$$

In steady-state simulation we are interested in the the performance measure $\mathbb{E}[Y^{\rm st}]$. In the examples above this means:

- $\mathbb{E}[Y^{\rm st}] = L$ = the expected number of customers in steady-state;
- $\mathbb{E}[Y^{\rm st}] = \mathbb{P}(W_q^{\rm st} > 0)$ = the probability of waiting in steady-state;
- $\mathbb{E}[Y^{\rm st}] = W_q$ = the expected waiting time of a customer in steady-state;
- $\mathbb{E}[Y^{\rm st}] = \mathbb{P}(W_q^{\rm st} \leq A_w)$ = the probability of waiting at most A_w in steady-state;
- $\mathbb{E}[Y^{\rm st}] = \mathbb{P}(\text{busy})$ the probability that the server in steady-state is busy;

These are steady-state performance measures, and we say that we wish to estimate the expectation of <`whatever`> in steady-state. The problem is how we can estimate these expectations of limit (or steady-state) random variables.

11.1 Simulation Run

Suppose that in your system (i) $X(t) \xrightarrow{\rm D} X$ (convergence in distribution), and that the function H is sufficiently proper so that also (ii) $H\big(X(t)\big) \xrightarrow{\rm D} H(X) = Y^{\rm st}$. Furthermore, suppose that (iii) $H\big(X(t)\big) \xrightarrow{\rm L_1} H(X)$ (convergence in expectation). In queueing models, typically all these hold.

The object to compute (estimate) is

$$\mu \doteq \mathbb{E}[Y^{\rm st}] = \mathbb{E}\big[H(X)\big].$$

Now reason that from (i) it follows that at a sufficiently large time τ_0 the system is about in steady-state, and thus it stays in steady-state; i.e. $X(t) \overset{\rm D}{\approx} X$ for all $t \geq \tau_0$. Then (ii) and (iii) say that $H\big(X(t)\big) \overset{\rm D}{\approx} H(X)$ for all $t \geq \tau_0$, and

$$\mu = \mathbb{E}\big[H\big(X(t)\big)\big] \approx \mathbb{E}\big[H(X)\big],$$

for all $t \geq \tau_0$.

Exploit these properties for a simulation run. You start simulating a run from an arbitrary initial state, for instance the empty state. Continue long enough, say until τ_0, at which the system is about in steady-state ($X(\tau_0) \overset{\rm D}{\approx} X$). Then continue simulating until $\tau_0 + \tau$, and define as output of the run

$$Y \doteq \frac{1}{\tau} \int_{\tau_0}^{\tau_0+\tau} H(\big(X(t)\big)\, dt.$$

Note that this is for continuous-time processes. Reason simularly for discrete-time processes, for instance the waiting times of the consecutive customers, and define sum-estimators. The output Y satisfies

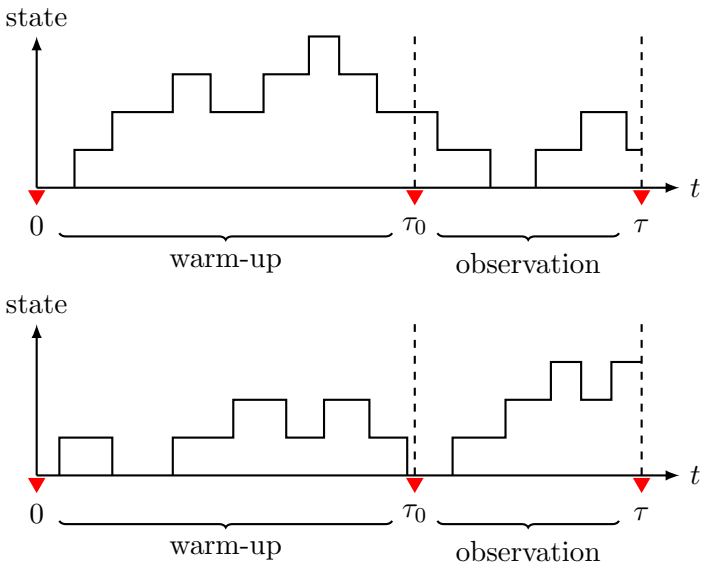
$$\begin{aligned}\mathbb{E}[Y] &= \mathbb{E}\Big[\frac{1}{\tau} \int_{\tau_0}^{\tau_0+\tau} H(\big(X(t)\big)\, dt\Big] \\ &= \frac{1}{\tau} \int_{\tau_0}^{\tau_0+\tau} \mathbb{E}\big[H\big(X(t)\big)\big]\, dt \approx \mu.\end{aligned}$$

11.2 Replication-Deletion

Repeat n times independently the simulation of a run described above. Then you get an (approximately) unbiased sample average estimator of μ by

$$\overline{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

Analyzing the estimator for variance goes as in Section 10. This method of steady-state simulation is called replication-deletion. The time interval $[0, \tau_0]$ is called the warm-up period which brings the system in steady-state (aka equilibrium), see Figure 6. The warm-up period is not used for computing output.



run 1: compute output Y_1 from state process in $[\tau_0, \tau_0 + \tau]$.

run 2: compute output Y_2 from state process in $[\tau_0, \tau_0 + \tau]$.

Figure 6: Illustration of 2 steady-state simulation runs.

11.3 Batch-Means

The replication-deletion has the advantage that it produced IID outputs $Y_1, \ldots, Y_n$, and thus the usual statistics are applicable. Disadvantage is its inefficiency because each run has a (large) warm-up period which is not used for computating the estimator. A remedy is the method of batch-means. In this method you simulate a single long run, say of length T. It consists of a first part of a warm-up period of length τ_0. Then you split the remaining time $T - \tau_0$ in n sections (called batches) of length $\tau = (T - \tau_0)/n$, see Figure 7. Compute the sample mean of the i-th batch $(i = 1, \ldots, n)$

$$Y_i = \frac{1}{\tau}\int_0^{\tau} H\big(X(\tau_0 + (i-1)\tau + t)\big)\, dt.$$

Then, the final estimator is the average of these batch means

$$\overline{Y}_n = \frac{1}{n}\sum_{i=1}^{n} Y_i.$$

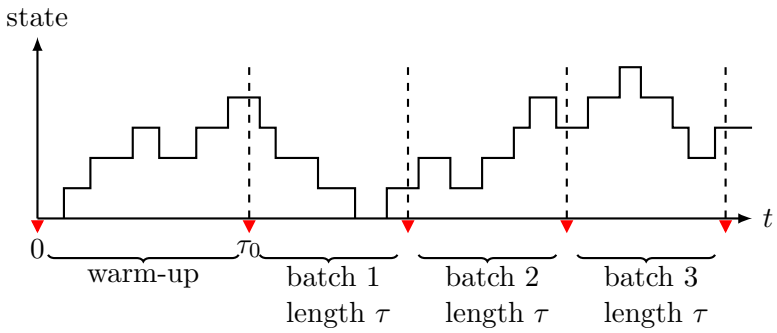


Figure 7: Illustration of a batch-means run.

Typically, the number of batches n is small. The batch length (or size) τ should be large, so that

1. The correlations between the batch means $Y_i, i = 1, 2, \ldots$ are neglegible.

2. Each batch mean Y_i (as an average) is approximately Gaussian.

Thus the Y_i's may be considered as independent Gaussian, with mean μ. The other statistics, such as variance, are computed as described in Section **??**.

11.3.1 Long-run Averages

Quite often one is interested in performances measures that are expressed os long-run averages, such as

- The long-run fraction of customers who leave unsatisfied.
- The long-run fraction of time that the machine is down.
- The long-run fraction of time the server is available.
- The long-run average cost per unit of time.

It is easy to see that these performance measures can be modelled by

$$\lim_{t\to\infty} \frac{1}{t} \int_0^t H\big(X(s)\big)\, ds,$$

for the appropiate function H. If the system state process $\{X(t), t \geq 0\}$ is regenerative these long-run average limits converge almost surely to the steady-state expectation (this is known as an ergodic theorem):

$$\lim_{t\to\infty} \underbrace{\frac{1}{t} \int_0^t H\big(X(s)\big)\, ds}_{\text{random variable}} = \mathbb{E}\big[H\big(X(\infty)\big)\big] \quad \text{a.s.}$$

For instance, for Markov chains this is relatively easy to prove. Regenerative processes are slightly more general. The main consequence is that you estimate long-run averages/fractions similarly as estimating $\mathbb{E}\big[H\big(X(\infty)\big)\big]$.

11.3.2 Remaining Issues

- The first question one may ask when executing replication-deletion, or batch-means is, how large should the length of warm-up period τ_0 be before starting the observations? This is called the initial transient problem. There are statistical techniques, and visiual techniques available. Here we advise to just simulate long enough and experiment.

- Another issue is that usually we are interested in two or more performance measures, for instance, the steady-state average waiting time, and the long-run fraction of time the server is busy. Then in each run or batch we make the observations for both outputs (we do not execute a separate simulation for each performance measure). When computing the associated outputs you need to take care of this.

12 Uniform Random Number Generating

Stochastic simulation is build upon the availability of random independent uniform (0,1) numbers $U_1, U_2, \ldots$. By the inverse transform method (or other algorithms) these uniform numbers are transformed to random numbers with other distributions, and these are used in stochastic processes.

For several reasons (see below), you do not wish true random numbers in computer simulation. There are algorithms that produce so-called pseudo-random numbers $u_1, u_2, \ldots$ that can be considered to be a realisation of the (perfect) $U_1, U_2, \ldots$. Such an algorithm is called *random number generator* (RNG). There are many RNG's. A good RNG satisfies the following criteria.

1. The produced sequence of numbers $u_1, u_2, \ldots$ passes successfully statistical tests on uniformity and independence.

2. The algorithm is very fast, e.g., 10M numbers in one second.

3. The sequence should be repeatable!

4. The algorithm is portable (works the same on different computers and operating systems).

5. The period of the sequence should be large. Explanation: computers are finite machines, thus at the end any (mathematically) infinitely long sequence repeats itself after what is called *the period*.

The default RNG in the module `random` of Numpy is the *PCG-64* which satisfies all these criteria with a period of 2^{128}.

The first number that is produced by the RNG depends on how the RNG is started. This is defined by an integer, called *seed*. Given the same seed, you get the same sequence of random numbers. Python's RNG starts up with a fresh unpredictable entropy pulled from the operating system. Thus, you need to set the seed by your self when you wish to reproduce the same sequence. In this way you can repeat a simulation experiment with the same sequence of random numbers! This is very important in scenario analysis, design optimization, etc. Here is an example.

```
import numpy.random as rnd

seed = 745795 # just any positive integer will do
rng = rnd.default_rng(seed)
```

```
# do an experiment, e.g.
x = rng.random(100)<0.5 # 100 tosses with fair coin

rng = rnd.default_rng(seed)
y = rng.random(100)<0.6 # 100 tosses with unfair coin
```

The 100 outcomes of the fair and unfair coins used the same random numbers. Thus for sure, if $x_i = 1$, then $y_i = 1$.

12.1 Random Data and Distributions

Using the random number generator you can sample from some well known distributions, check https://numpy.org/doc/stable/reference/random/generator.html

Table 1: Some available functions to generate random numbers.

function	description
`random`	uniform in $[0, 1)$
`integers`	random integers
`choice`	from a discrete distribution
`standard_normal`	from the standard Gaussian distribution
`binomial`	from binomial distribution
`gamma`	from Gamma distribution

Here are examples how to use these.

```
rng = rnd.default_rng()
u = rng.random() # one (0,1)-random number
u = rng.random(4) # array of four (0,1)-random numbers
u = rng.random([4,5]) # 4 x 5 matrix of (0,1)-random numbers
u = rng.random([4,5,3,3]) # 4 x 5 x 3 x 3 matrix
```

```
rng = rnd.default_rng()
x = rng.integers(5) # random in {0,1,2,3,4}
x = rng.integers(3,8) # random in {3,4,5,6,7}
x = rng.integers(3,8,size=10) # 10 times random in {3,4,5,6,7}
x = rng.integers(7,size=(4,5,3,3)) # 4 x 5 x 3 x 3 matrix
                    # of random integers in {0,1,...,6}
```

```
rng = rnd.default_rng()
z = rng.standard_normal() # one random Gaussian number
z = rng.standard_normal(4) # array of four random Gaussian numbers
z = rng.standard_normal([4,5]) # 4 x 5 matrix of random Gaussian numbers
z = rng.standard_normal([4,5,3,3]) # 4 x 5 x 3 x 3 matrix
```

12.2 Random Sample from a Finite Distribution

Assume that a finite distribution (say of size n) is given on state space $S = \{s_1, \ldots, s_n\}$ by a probabilities $p_1, \ldots, p_n$. Suppose that you want to generate m i.i.d. samples from this distribution; i.e., with replacement. The method for this is `choice()` which you call with arguments s, m, p. Example (as showing how to call another RNG)

```
prob = np.array([1,2,3,4,5,5,4,3,2,1])
prob = prob / np.sum(prob)
n = len(prob)
s = np.linspace(-np.pi,np.pi,n)
m = 100
rng = rnd.Generator_rng(rnd.MT19937) # Mersenne Twister
x = rng.choice(s, size=m, p=prob)
```

When you do not specify p, the samples are generated according to the uniform distribution on S. When the state space is $S = \{0, 1, \ldots, n-1\}$ it suffices to just give the size n:

```
prob = np.array([1,2,3,4,5,5,4,3,2,1])
prob = prob / np.sum(prob)
n = len(prob)
m = 100
x = rng.choice(n, size=m, p=prob)
```

You generates samples without replacement by the argument `replace=False`:

```
prob = np.array([1,2,3,4,5,5,4,3,2,1])
prob = prob / np.sum(prob)
n = len(prob)
m = 5
x = rng.choice(n, size=m, replace=False, p=prob)
```

12.3 Legacy Random Generation

The class `RandomState` is NumPy's legacy generator based on the Mersenne Twister *MT19937* which is considered to be slow in comparison to *PCG-64*. It remains available but will not get improvements. It has the same functionalities as the newer the `Generator` class. Seeding and calls to random uniforms, integers, standard normals, finite distributions goes as follows.

```
import numpy as np
import numpy.random as rnd

seed = 12345
rnd.seed(seed) # sets the seed of the MT to <seed>

u = rnd.rand() # single
u = rnd.rand(100) # array of size 100
j = rnd.randint(5) # random in {0,1,2,3,4}
j = rnd.randint(3,8) # random in {3,4,5,6,7}
j = rnd.randint(3,8,size=10) # 10 times random in {3,4,5,6,7}
z = rnd.randn() # single
z = rnd.randn(2000) # array of size 2000
x = rnd.choice(np.array([-1,0,1]), size=5, p=np.array([0.2,0.3,0.5]))
# 5 times with replacements from {-1,0,1} according to p
```

13 Nonuniform Random Number Generating

How to generate a sequence of numbers $x_0, x_1, \ldots$ such that it is a realisation of a sequence of IID (independent, identically distributed) random variables $X_0, X_1, \ldots$? Let X be the generic

random variable, with cumulative probability function (CDF) $F(x)$, and probability density function (PDF) $f(x)$. Assuming that the RNG produces IID uniform random numbers, it suffices to consider algorithms for generating a single random sample (or realisation or value or observation) of X. Also, one says to be able to simulate from F (of f).

13.1 Inverse Transform Method

Define for $u \in (0,1)$,

$$F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}.$$

It is simple to show that

$$U \stackrel{\mathrm{D}}{=} \text{Uniform}(0,1) \quad \Rightarrow \quad F^{-1}(U) \stackrel{\mathrm{D}}{=} X,$$

i.e. $\mathbb{P}(F^{-1}(U) \leq x) = F(x)$ for all $x \in \mathbb{R}$. Thus the inverse transform algorithm is:

1. generate a pseudo-random number u in $(0,1)$;
2. return $x = F^{-1}(u)$.

Notice that the inverse transform method is a perfect (or unbiased) method. Unfortunately, most continuous CDF's cannot be inverted analytically; a numerical approximation would generate biased samples. Discrete random variables can always be generated by the inverse transform method: let $(p_j)_{j=0}^{\infty}$ be the PDF of X, then clearly

$$X = k \quad \Leftrightarrow \quad \sum_{j=0}^{k-1} p_j < U \leq \sum_{j=0}^{k} p_j.$$

The inverse transform algorithm for discrete random variables is:

1. $j = 0$; $s = p_0$;
2. generate a pseudo-random number u in $(0,1)$;
3. while $u > s$, do $j = j + 1, s = s + p_j$, end;
4. return j.

This becomes inefficient when X has small (positive) probabilities on a large support.

13.2 Acceptance-Rejection Method

Here we explain it for the continuous case; for the discrete case you can adapt accordingly. Let $f(x)$ be the PDF of X on $\mathbb{R}$, and suppose that there is an integrable function t that majorises f; that is $t(x) \geq f(x)$ for all $x \in \mathbb{R}$, and $\int_{-\infty}^{\infty} t(x)\,dx = c < \infty$. Then define $g(x) = t(x)/c$. Notice, $g(x) \geq 0$ for all $x \in \mathbb{R}$, and $\int_{-\infty}^{\infty} g(x)\,dx = 1$. Thus, g is a PDF, say of a random variable Y. Now you can prove that

$$Y | (U \leq f(Y)/t(Y)) \stackrel{\mathrm{D}}{=} X; \quad \text{where } U \stackrel{\mathrm{D}}{=} \text{Uniform}(0,1).$$

That is, for all $x \in \mathbb{R}$,

$$\mathbb{P}\Big(Y \leq x \Big| U \leq \frac{f(Y)}{cg(Y)}\Big) = \mathbb{P}(X \leq x).$$

Thus the acceptance-rejection algorithm is:

1. generate a realisation y from Y using PDF g;

2. generate a pseudo-random number u in $(0, 1)$;

3. if $u \leq f(y)/(cg(y))$ return y; else repeat from step 1.

Notice that the acceptance-rejection method is a perfect (or unbiased) method. Notice also that the number of iterations before acceptance is a random variable (when the algorithm is randomised) with a geometric distribution with success probability the probability of acceptance

$$p = \mathbb{P}\Big(U \leq \frac{f(Y)}{cg(Y)}\Big) = \frac{1}{c}.$$

Thus the objective is to find a density g for which $c \geq 1$ is as small as possible. Many well-known distributions are simulated by the acceptance-rejection method.

13.3 Convolution

Suppose that $X = \sum_{k=1}^{n} Y_k$ where $Y_1, \ldots, Y_n$ are independent random variables with CDF's $G_k(y), k = 1, \ldots, n$. The convolution algorithm is

1. for $k = 1, \ldots, n$, generate a realisation y_k from Y_k using CDF G_k;

2. return $x = \sum_{k=1}^{n} y_k$.

13.4 Composition

Suppose that the CDF of X is a mixture of CDF's:

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

where $p_j \geq 0$, $\sum_{j=1}^{\infty} p_j = 1$. Each function G_j is the CDF of some random variable Y_j. Let J be a discrete random variable on $\{1, 2, \ldots\}$ with PDF $(p_j)_{j=1}^{\infty}$. Then

$$X \stackrel{\mathrm{D}}{=} \sum_{j=1}^{\infty} Y_j 1\{J = j\}.$$

The composition algorithm is

1. generate a realisation k from J using PDF (p_j)

2. generate a realisation y_k from Y_k using CDF G_k;

3. return $x = y_k$.

13.5 Special Distributions in Python

For many special distributions such as Gamma, Weibull, Lognormal, etc, you find a sampling function in the module `scipy.stats`. For instance sampling 1000 times from the Gamma distribution with shape parameter $\alpha = 2.4$ and scale parameter $\theta = 0.7$, i.e., pdf is

$$f(x) = \frac{x^{\alpha-1}}{\theta^{\alpha}\Gamma(\alpha)}\, e^{-x/\theta}, \quad x \geq 0.$$

Python command

```
alpha = 2.4
theta = 0.7
n = 1000
x = scipy.gamma.rvs(alpha, scale = theta, size = n)
```

14 Simulating Poisson processes

14.1 Homogeneous Poisson Process

A Poisson process $\{N(t), t \geq 0\}$ is most easily simulated by applying the property that the interarrival times $X_1, X_2, \ldots$ are i.i.d. exponentially distributed random variables with mean $1/\lambda$. Thus, to generate a sequence $S_0 = 0, S_1, S_2, \ldots$ of consecutive arrival epochs, you set $S \leftarrow 0$, and repeat

Step 1 : draw a sample X from the exponential(λ)-distribution;

Step 2 : $S \leftarrow S + X$

14.2 Nonhomogeneous Poisson Process

Let $\lambda : [0, \infty) \to [0, \infty)$ be a nonnegative function on the time-axis. A nonhomogeneous Poisson process with rate function λ is a counting or point or pure jump process $N(t), t \geq 0$ with the properties

1. $N(0) = 0$, and nonnegative integer increments $N(t) - N(s) \in \{0, 1, \ldots\}$ for all $0 \leq s < t$;
2. it has independent increments, i.e., $N(t_1) - N(s_1)$ and $N(t_2) - N(s_2)$ are independent for all $0 \leq s_1 < t_1 < s_2 < t_2$;
3. $\mathbb{P}(N(t+h) - N(t) = 1) = h\lambda(t) + o(h),\ h \to 0$;
4. $\mathbb{P}(N(t+h) - N(t) \geq 2) = o(h),\ h \to 0$.

Denote S_k for the k-the jump time of the process: $S_0 = 0$, and

$$S_{k+1} = \inf\{t > S_k : N(t) = N(S_k) + 1\}.$$

In a discrete event simulation of a queue with nonhomogeneous Poisson arrival process you need to simulate the next arrival time $S_{k+1} = S_k + A$ when the current time S_k is an event time of an arrival. Here is an algorithm assuming that the rate function λ is bounded: $\lambda(t) \leq \bar{\lambda} < \infty$ for all $t \geq 0$.

Step 0 : given current time S of an arrival; set $T = S$;

Step 1 : draw a sample X from the exponential($\bar{\lambda}$)-distribution; set $T = T + X$;

Step 2 : draw a sample U from the uniform$(0, 1)$-distribution; if $U < \lambda(T)/\bar{\lambda}$ return $S_{k+1} = T$ (interarrival time is $A = T - S_k$) and stop, else repeat from Step 1.

This algorithm is a so-called thinning procedure of the (homogeneneous) Poisson-$\bar{\lambda}$ process.

15 Common Random Numbers

The technique of *common random numbers* (CRN) is a method that we apply when we estimate performance measures under various scenario's, designs, or model configurations. It is often applied in practice because it is natural to compare alternative systems under the same random circumstances. The idea is that you use the same stream of uniform random numbers for generating the same random variable in the different systems that you are comparing.

When implementating CRN you should take into account that *the same* random variables are simulated by *the same* random uniform numbers from the RNG.

Example: the manager of a small shop considers to replace two slow cashiers by one which is twice as fast. What will happen with the service quality, i.e., the average waiting time per customer. In other words, the problem is to compare $\mathbb{E}[W]$ in $G/G/1$ and $G/G/2$ queueing models, where

- W customer's waiting time (transient or steady-state);
- equal arrival processes in both systems ('the same customers');
- equal job requirements in both systems ('the same items bought');
- the procesing times differ a factor 2.

Suppose that you generate consecutive interarrival times $A_1, A_2, \ldots$ for the $G/G/1$ simulation. Synchronisation means that you use these same interarrival times for the $G/G/2$ simulation.

Analogously for the associated job requirements $S_1, S_2, \ldots$ of customers $1, 2, \ldots$. For this kind of re-use of random numbers one may utilise an RNG with streams.

16 Verification

Verification is the process of determining whether the model has been correctly translated into a computer program. This is also called the debugging of your program. Techniques include:

- Check whether the random variable generations are correct: make histograms; simulate an average and compare with theoretical mean; similar for the variance.
- Print a trace: eventlist, state, statistics after each event during a small portion of a simulation run.
- Check properties such as 'Little'.
- Simulate a version for which theoretical values of the performance measures are availabe.
- Simulate with extreme or boundary values of the parameters.

References

S. Asmussen and P.W. Glynn, *Stochastic Simulation. Algorithms and Analysis*, Springer 2007.

C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*, Second edition, Springer 2009.

G. S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*, Springer 2001.

D.P. Kroese, T. Taimre, Z.I. Botev, *Handbook of Monte Carlo Methods*, Wiley 2011.

A. M. Law. *Simulation Modeling and Analysis*, McGraw-Hill fifth edition 2015.

B.L. Nelson. *Foundations and Methods of Stochastic Simulation*, Springer 2013.

H. Perros. *Computer Simulation Techniques: The definitive introduction!*, 2009, available from https://people.engr.ncsu.edu/hp/files/simulation.pdf

B.D. Ripley, *Stochastic Simulation*, Wiley 1987.

R.Y. Rubinstein and D.P. Kroese. *Simulation and the Monte Carlo Method*, Wiley third edition 2017.