Discrete-event simulation

Hand-out based on Section 10.2 of the book
'Operationele analyse' by Prof. dr. H.C. Tijms

We have seen how to generate random numbers from many different probability distributions. We have also seen how to apply this knowledge to e.g. find the stationary expected number of customers in a G/G/1 queue and the time till ruin in the Cramér-Lundberg and Sparre-Andersen risk models. These measures exhibited extraordinarily nice expressions which allowed a straightforward simulation (for the G/G/1: Lindley's recurrence relations). In general, however, we lack these devices. In this case, the method of discrete-event simulation can be applied. The principle of this method is perhaps best explained through a concrete example.

Example. A bank opens at 10AM. There are c bank clerks. Each of them are equally fast and are able to serve any customer. A careful data analysis shows that during opening hours customers arrive at the bank according to a Poisson process with an arrival rate of λ per minute. In other words, the interarrival times of customers are mutually independent and exponentially distributed with parameter λ. The arriving customers line up in a single queue. When a bank clerk becomes available, the front customer of the queue will be served by this bank clerk. The service times of these customers are mutually independent random variables and are in addition independent from the arrival process. The aforementioned data analysis shows that the services times of all customers are each uniformly distributed between a minutes and b minutes. At 5PM, the bank closes its doors, but the clerks will still serve the remaining customers in the queue until the queue is empty.

Problem statement

Determine for given values of c, λ, a and b:

a) the average length of the queue during the day

b) the average waiting time of a customer

c) the fraction of time that the clerk is busy.

This problem is known in the literature as the M/G/c queueing problem. No analytical results for this problem exist in the literature. Therefore, we will have to resort to simulation. We will apply the method of discrete event simulation. To this end, we will first define the notion of states, events and statistical counters.

States

The key to building a good model is the choice of all possible states of the system. The state of a system must contain enough information about the history of the system such

that, given the current state, the previous states become irrelevant to predict the future behaviour of the system. The choice of which variables to include in the state of the system is heavily influenced by the system quantities one wants to investigate.

In the example used in this document, the state variables are as follows:

- the status of the bank clerks (busy or not)
- the number of customers in the queue
- the arrival times of the customers in the queue (necessary to determine the average waiting time of a customer)

Note that the same state variables would be used if we would have assumed other distributions for the interarrival and service time distributions.

Events

Another important notion in discrete event simulation encompasses events, which can potentially change the state of the model. For discrete event simulation, it is essential that events can only occur at discrete point in time, so that the state of the model will not continuously change over time. In our bank example, the events will be the arrivals of customers and the departures of customers.

The assumption that events can only occur discretely in time, enables us to compress the true scale of time into just changing the 'simulation clock' only at moments at which an event occurs. At these moment, the state of the system changes. The subsequent steps of the simulation clock are usually of unequal duration. In a discrete event simulation, time intervals where no relevant changes to the system occur, and thus where the state of the system does not change, are *skipped*. With that, unnecessary simulation efforts are prevented.

Statistical counters

Statistical counters are variables that store statistical information which can be used to compute the requested performance measures at the end of a simulation. In the bank example we will use the following statistical counters:

- the number of customers that arrived in the system so far
- the total waiting time of customers that have arrived so far (the waiting time of a customer does not include the actual service time)
- total amount of time that a bank clerk has been busy so far
- total surface under the graph of the number of customers in the queue so far (this number does not include the customers being served by a bank clerk)

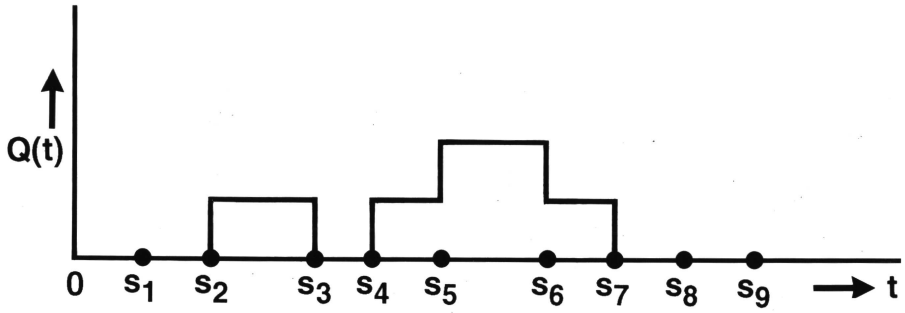


Figure 1: Flow chart of the program simulating the M/G/c bank model

The statistical counters are not kept up to date continuously. Instead, just as is the case with the state variables, the statistical counters are only updated at the (discrete) points in time at which an event occurs. The last of the above-mentioned statistical counters may need some explanation. Let us define $Q(t)$ to be the number of customers in the queue at time t. This is a step function that can only change its value when an event (either arrival or departure) occurs. If we assume that the number of bank clerks equals $c = 1$, then the graph in Figure 1 could occur. In this graph, s_i is the occurence time of the i-th event. In particular, in this realisation s_1, s_2, s_4, s_5 and s_9 are arrival epochs and s_3, s_6, s_7 and s_8 are departure epochs. The average number of customers in the queue after a simulation of T time units can now be estimated by

$$\hat{Q} = \frac{1}{T} \int_0^T Q(t)dt.$$

In the example of the bank clerks, we will take T equal to the length of the period the bank is open, i.e. the time between 10AM and 5PM. Although the integrand is a continuous time function, for computation purposes it is enough to do the bookkeeping only when events occur. The integral can simply be computed by summing the surfaces of a number of rectangles. The surfaces of the same rectangles enable us also to compute the waiting time per customer. The total surface under the graph up till time T divided by T equals the average length of the queue, while the average waiting time per customer follows by dividing the total surface under the graph until time T by the number of customers that arrived until time T. The latter computation has a big connection with the law of Little known from queueing theory.

It is important to note that after a single simulation run, the value that we found for the average queue length, average waiting time or the fraction of time that a bank clerk is busy is a realisation of a random variable. As such, the found value may change significantly per simulation run. The fluctuations that arise may be significant, especially when the queueing system is heavily loaded. It is because of these fluctuations that one may not limit himself to a simulation that consists of a single observation. One needs a large number of observations to average out these fluctuations. A careful statistical output analysis will therefore be of an utmost importance to come to useful conclusions. We will

come back to this point later in this course. For now, we will focus on how to perform a simulation run by using the computer.

Idea behind computer program

The question now is how a good computer program would look like. Above, we have already noted that the idea around which the simulation scheme revolves is: *the simulation clock skips the time interval in between two successive events as being irrelevant.* This happens, since in reality no event (arrival, service completion, etc.) occurs in between the two time epochs. This means that in the simulation program we only regard the time epochs at which an event occurs. At those time epochs the state variables and the statistical counters (the bookkeeping) will be adjusted. So, in the simulation program, we constantly have to look for the next event. The activation of an event typically triggers the scheduling of future related events. In the program of our bank example, for instance, the handling of an arrival event will also trigger the scheduling of the arrival of the next customer, which requires a random number from the inter-arrival time distribution. In addition, if a bank clerk is available at the time corresponding to the arrival event being handled, the departure of the customer can be scheduled as well by generating a random number from the service time distribution.

In the simulation program, an event list is used. An event list is a list of events that are already scheduled with their corresponding time of occurrence. It is important that this list be managed effectively. If the size of the event list (i.e. the number of events in the event list) is generally not too large, a data structure like an array in Mathematica will do. Alternatively, a linked list can be used in many program languages, as the events in such a data structure are sorted automatically in the order of time of occurence. If the event list, however, is typically very large, other data structures are needed, such as binary trees or heaps. A detailed treatise on data structures is however beyond the scope of this course.

All of the above becomes clear by regarding a concrete example. To this end, Figure 2 gives a flow chart of the M/G/c bank model that shows schematically the steps that are taken in the Mathematica program. In the simulation program, one day of seven hours is simulated with $c = 3$ bank clerks. On average, one customer arrives every minute and each bank clerk should be able to help 30 customers per hour on average if they were constantly busy. In particular, the distribution of the service time that each customer requires is uniformly distributed on $(0, \frac{1}{15})$; time unit: hours. In the sequel, we present the code of a simulation program that is written for the bank example in Mathematica. The usual rules for correct programming have been followed. This means first and foremost: *modularise!* So, a separate module for each event routine, each procedure for drawing random numbers etc. For didactic purposes, this program limits itself to a single simulation run of the bank model. To draw reliable conclusions concerning the behaviour of the bank model, the program of course has to be adjusted so that many runs are possible, and the program needs to be extended with modules for the *output analysis* of several runs. This topic will be treated in an upcoming lecture.

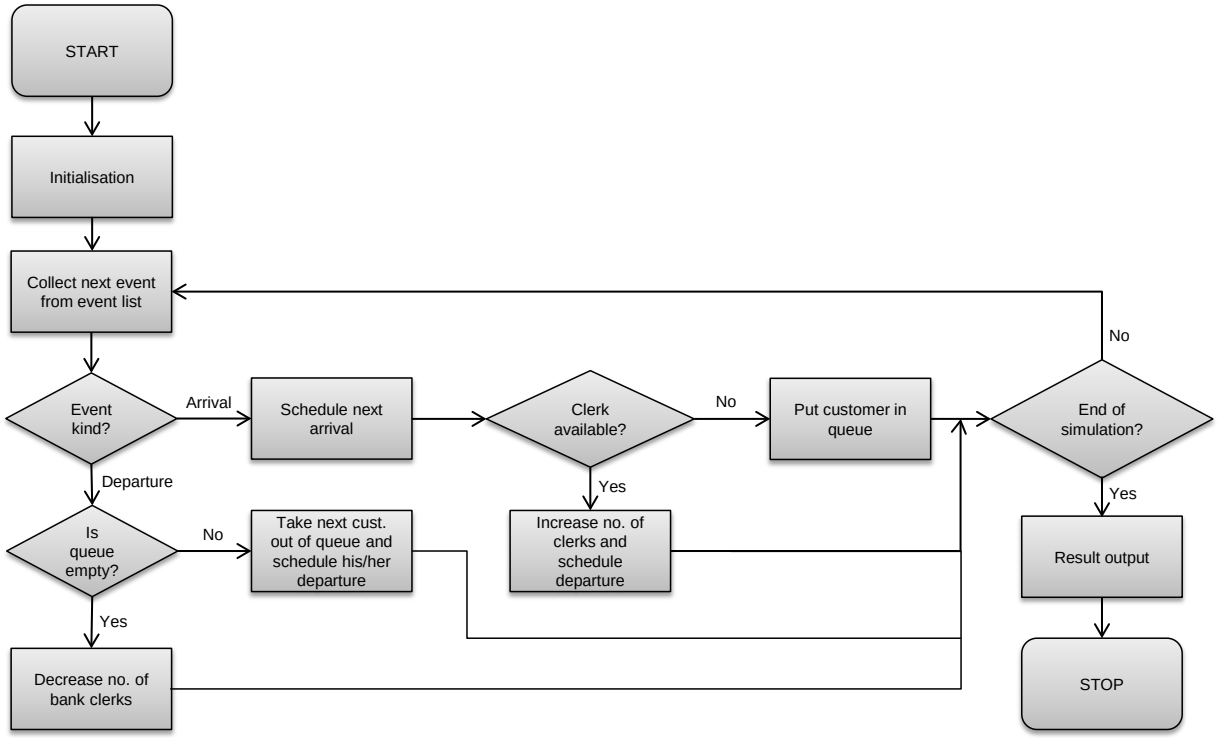


Figure 2: Flow chart of the program simulating the M/G/c bank model

Program in Mathematica

sim_bank_example.nb *

M/G/c bank example
Mathematica code accompanying handout for the course *Stochastic Simulation*
Jan-Pieter Dorsman / April 15 2017

In[15]:
```
(* Function that defines parameters and performs preliminary work *)
preliminaries[] := Module[{},
                          (* Model parameters *)
                          c = 3; (* Number of bank clerks *)
                          λ = 60; (* Arrival rate *)
                          lowerParamServiceTime = 0; (* The lower bound on the uniform service time *)
                          upperParamServiceTime = 1/15; (* The upper bound on the uniform service time *)
                          numberOfHoursInADay = 7; (* The number of hours to be simulated *)

                          (* Definitions *)
                          KINDARRIVAL = 1;
                          KINDDEPARTURE = 2;

                          (* Statistical Counters *)
                          totalNumberOfArrivals = 0;
                          totalWaitingTimeOfCustomers = 0;
                          totalAmountOfTimeClerkIsBusy = 0;
                          totalSurfaceUnderGraph = 0; (* counter used for the average queue length *)

                          (* Other parameters *)
                          eventList = {}; (* Events in the list have the form {timeOfOccurence, kind}, e.g. {5.2857, KINDDEPARTURE} *)
                          queue = {}; (* Customers in the queuehave the form {time of arrival}, e.g. {3.4874} *)
                          simulationClock = 0;

                          currentNumberOfClerksBusy = 0;

                          SeedRandom[123]; (* Reset the random number generator and set its seed equal 12345 *)
                         ];
```

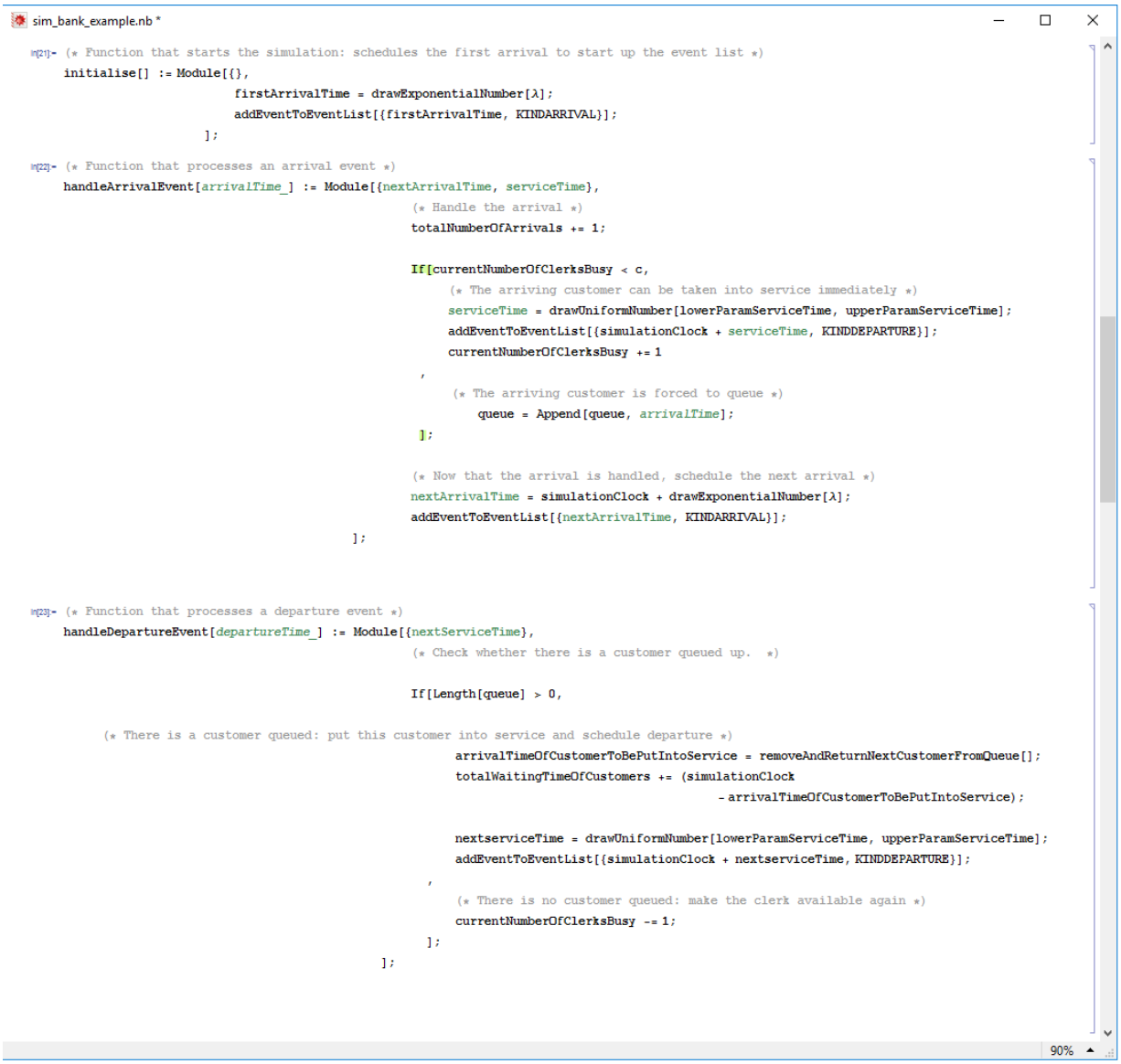
In[16]:
```
(* Function that draws a number from the uniform distribution on [a,b],
using a standard uniform sample given by the function RandomReal[] *)
drawUniformNumber[a_, b_] := a + (b - a) * RandomReal[];
```

In[17]:
```
(* Function that draws a number from the exponential
 ( distribution, using a standard uniform sample given by the function RandomReal[] *)
drawExponentialNumber[rate_] := -Log[RandomReal[]]/rate;
```

In[18]:
```
(* Function that adds an event to the event list *)
addEventToEventList[event_] := Module[{},
                                      eventList = Sort[Append[eventList, event]];
                                     ];
```

In[19]:
```
(* Function that removes the next event from the event list and returns it *)
removeAndReturnNextEventFromList[] := Module[{nextEvent},
                                             nextEvent = First[eventList];
                                             eventList = Drop[eventList, 1];
                                             nextEvent
                                            ];

(* Function that removes the next customer from the queue and returns it *)
removeAndReturnNextCustomerFromQueue[] := Module[{nextCustomer},
                                                 nextCustomer = First[queue];
                                                 queue = Drop[queue, 1];
                                                 nextCustomer
                                                ];
```

90%


sim_bank_example.nb *
In[21]:= (* Function that starts the simulation: schedules the first arrival to start up the event list *)
initialise[] := Module[{},
firstArrivalTime = drawExponentialNumber[λ];
addEventToEventList[{firstArrivalTime, KINDARRIVAL}];
];
In[22]:= (* Function that processes an arrival event *)
handleArrivalEvent[arrivalTime_] := Module[{nextArrivalTime, serviceTime},
(* Handle the arrival *)
totalNumberOfArrivals += 1;
If[currentNumberOfClerksBusy < c,
(* The arriving customer can be taken into service immediately *)
serviceTime = drawUniformNumber[lowerParamServiceTime, upperParamServiceTime];
addEventToEventList[{simulationClock + serviceTime, KINDDEPARTURE}];
currentNumberOfClerksBusy += 1
,
(* The arriving customer is forced to queue *)
queue = Append[queue, arrivalTime];
];
(* Now that the arrival is handled, schedule the next arrival *)
nextArrivalTime = simulationClock + drawExponentialNumber[λ];
addEventToEventList[{nextArrivalTime, KINDARRIVAL}];
];
In[23]:= (* Function that processes a departure event *)
handleDepartureEvent[departureTime_] := Module[{nextServiceTime},
(* Check whether there is a customer queued up. *)
If[Length[queue] > 0,
(* There is a customer queued: put this customer into service and schedule departure *)
arrivalTimeOfCustomerToBePutIntoService = removeAndReturnNextCustomerFromQueue[];
totalWaitingTimeOfCustomers += (simulationClock
- arrivalTimeOfCustomerToBePutIntoService);
nextserviceTime = drawUniformNumber[lowerParamServiceTime, upperParamServiceTime];
addEventToEventList[{simulationClock + nextserviceTime, KINDDEPARTURE}];
,
(* There is no customer queued: make the clerk available again *)
currentNumberOfClerksBusy -= 1;
];
];
90%

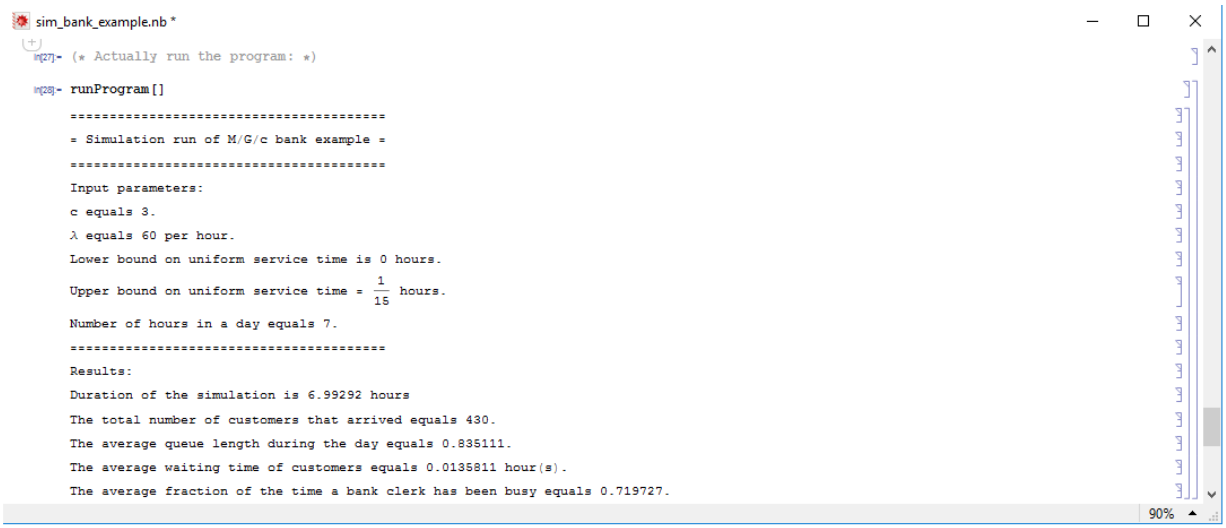
sim_bank_example.nb *

In[24]:
```
(* Function that takes the next event from the event list and processes it. *)
handleNextEvent[] := Module[{nextEvent, eventTime, eventKind},
                                nextEvent = removeAndReturnNextEventFromList[];
                                eventTime = nextEvent[[1]];
                                eventKind = nextEvent[[2]];

                                (* Update counters and parameters *)
                                totalSurfaceUnderGraph += (eventTime - simulationClock) * Length[queue];
                                totalAmountOfTimeClerkIsBusy += (eventTime - simulationClock) * currentNumberOfClerksBusy;

                                simulationClock = eventTime;

                                (* Checks whether the next event is an arrival or a departure and calls functions accordingly *)
                                If[eventKind == KINDARRIVAL ,
                                     handleArrivalEvent[eventTime];
                                 ,
                                     handleDepartureEvent[eventTime];
                                 ];
                            ];
```

In[25]:
```
(* Function that reports the input parameters and the findings of the simulation program *)
report[] := Module[{},
                            Print["========================================"];
                            Print["= Simulation run of M/G/c bank example ="];
                            Print["========================================"];
                            Print["Input parameters:"];
                            Print["c equals ", c, "."];
                            Print["λ equals ", λ, " per hour."];
                            Print["Lower bound on uniform service time is ", lowerParamServiceTime, " hours."];
                            Print["Upper bound on uniform service time = ", upperParamServiceTime, " hours."];
                            Print["Number of hours in a day equals ", numberOfHoursInADay, "."];
                            Print["========================================"];
                            Print["Results:"];
                            Print["Duration of the simulation is ", simulationClock, " hours"];

                            Print["The total number of customers that arrived equals ", totalNumberOfArrivals , "."];
                            Print["The average queue length during the day equals ", totalSurfaceUnderGraph / simulationClock , "."];
                            Print["The average waiting time of customers equals ", totalWaitingTimeOfCustomers / totalNumberOfArrivals,
    " hour(s)."];
                            Print["The average fraction of the time a bank clerk has been busy equals ",
    totalAmountOfTimeClerkIsBusy / simulationClock / c, "."];

                ];
```

In[26]:
```
(* Main Function *)
runProgram[] := Module[{},
                                preliminaries[];
                                initialise[];

                                (* Handle events when the time of the earliest upcoming event, eventList[[1]][[1]],
   occurs before the end of the day, marked by numberOfHoursInADay *)
                                While[eventList[[1]][[1]] < numberOfHoursInADay,
                                        handleNextEvent[];
                                 ];

                                report[];
                ];
```

90%


sim_bank_example.nb *
In[27]:= (* Actually run the program: *)
In[28]:= runProgram[]
==========
= Simulation run of M/G/c bank example =
==========
Input parameters:
c equals 3.
λ equals 60 per hour.
Lower bound on uniform service time is 0 hours.
Upper bound on uniform service time = 1/15 hours.
Number of hours in a day equals 7.
==========
Results:
Duration of the simulation is 6.99292 hours
The total number of customers that arrived equals 430.
The average queue length during the day equals 0.835111.
The average waiting time of customers equals 0.0135811 hour(s).
The average fraction of the time a bank clerk has been busy equals 0.719727.
90%