TEACHING VARIANCE REDUCTION IN MATLAB

Jaroslav Sklenar

Department of Statistics and Operations Research
University of Malta
Msida MSD 2080, Malta

jaroslav.sklenar@um.edu.mt

ABSTRACT
The paper describes an application of a new tool for programmed discrete event simulation in Matlab that became one of the most frequently taught languages for computations in mathematics, statistics, and operations research. To teach simulation related topics we need a simple to use and a fast to learn tool for creation of simple and medium-scale simulation models. We also need a tool where the code is visible and accessible and where all functions like generation of random numbers are directly under user's control. All this is caused by the need to incorporate simulation models into various algorithms based on repetitive experiments, variance reduction techniques, and simulation-based optimization. After a short introduction of the tool we present variance reduction examples.

Keywords: discrete event simulation, queuing systems, variance reduction, matlab.

1. INTRODUCTION

It is a well-known fact that for classical simulation applications like manufacturing, transportation, or similar described typically as queuing systems, programming is used less and less. For these systems the classical GPSS view of the world as represented by interactive tools like Arena, Simul8, Witness, and similar is satisfactory and programming simulation models of such systems is often considered as a waste of time and money. Fortunately there are still areas where simulation techniques are becoming more and more important and where the classical view of entities passing through a block diagram does not work. This is true in stochastic programming, finance, stochastic integration, reinforcement learning to mention just a few. In our situation there are two more arguments in favor of programmed simulation models where the user has full control over the model. In Statistics and OR courses we have recently introduced a study-unit called "Computational Methods in Statistics and OR" for students who know only basics of programming in Matlab. For the simulation part of this unit the obvious choice was an interactive simulation tool, in our case Arena (Kelton et al. 2006). Problems started with teaching Variance Reduction Techniques (L'Ecuyer 2007). Though some of these techniques are included in Arena and similar packages, we need to show their implementation. Another area where full control over the model is required is simulation-based optimization. There are optimization tools included in interactive simulation tools like OptQuest of Arena (Bradley 2007), but there is no feasible possibility to apply other than the built-in optimization algorithm and control over its working is very limited, leaving alone techniques like for example infeasibility detected by simulation.

So to summarize, we need a simple to use and a simple to learn tool for creating discrete event simulation models in Matlab. Simulation models should take a form of a function that given model specification and run control arguments provides the required results as outputs. Such function can then be incorporated into other algorithms, in our case algorithms used in variance reduction and simulation-based optimization.

2. SIMULATION IN MATLAB

Support is needed for simulation models with continuous time and discrete behavior. Simulation of discrete time or timeless models typical in finance and stochastic programming (often called Monte Carlo simulation) is from the time control point of view relatively easy and no special support in Matlab is needed. We are aware of two Matlab based discrete event simulation tools. SimEvents (Gray 2007) is a commercial interactive tool based on Simulink of Matlab. It belongs to the category of interactive tools with limited control over the model. MatlabDEVS2 (Deatcu 2003) is a tool created primarily as a support for research and education of abstract DEVS theory, so its use is not practical in our case either. That's why it has been decided to create a new tool with simplicity and transparency being the main objectives. The tool is definitely not supposed to be used for large-scale computationally demanding simulation studies.

3. TOOL DESCRIPTION

The paper (Sklenar 2013) describes the ideas and the implementation details of the tool. Here we just summarize its functions by categories.

3.1. Time Control Functions

The tool uses the classical event-oriented paradigm based on the sequencing set made of event notices. In our case the event notices are made of the event time, unique event notice identification, user event number, and user event notice data. The sequencing set is made of four arrays whose *i*-th items represent the event notice *i*. The set is not ordered, scheduling places the new items at the end, next event to be activated is found by the function `min` of Matlab in the array of event times. Removing notices is done in usual Matlab way by storing empty values [ ] in the four items. This approach is certainly not very fast, but it is simple and it works satisfactorily. As the system code is not protected, we start all system identifiers by "`s_`".

`s_time` is the system variable that contains the current model time.

`function id = s_chedule(t,e,d)` schedules the event *e* at time *t* with user data *d*. It returns the event notice identification *id* assigned by the engine.

`function s_cancel(id)` removes the event notice *id* from the sequencing set.

`function s_imulation` starts the simulation run. It is assumed that at least one event has been scheduled.

`function s_terminate` ends the simulation run by clearing the sequencing set.

In addition to the above functions, the user has to write the common user event function:

`function event(e,d,id)` that starts the event *e* with data *d* and identification *id*. It typically tests the event number *e* and activates the particular event function. In addition to user events, there may be system events with negative numbers used by application-oriented additions to the basic tool - see later.

The simulation engine is the function `s_imulation` that repeatedly removes the next event notice from the sequencing set and activates either the user function `event` or a hidden system event function. The run ends when the empty sequencing set is detected.

3.2. Statistics

With respect to time there are two types of statistics. Time dependent statistics (using Arena's terminology *time-persistent statistics*) is based on time integrals. We call such statistical objects *accumulators*, typical example is the statistics on a queue length. The other type is statistics based only on a collection of assigned values (using Arena's terminology *counter statistics*). We call such statistical objects *tallies*, typical example is the statistics on waiting time in a queue. The following functions are available:

`function s_tupdate(t,x)` updates the tally *t* by the value *x*. The function keeps the minimum and the maximum values, the sum of assigned values, the sum of squared assigned values and the number of updates.

`function [mean,min,max,variance,updates] = s_tallystat(t)` returns the descriptive statistics on the tally *t*.

`function s_aupdateto(a,x)` updates accumulator *a* to the value *x*. Call to this function replaces the assignment $a = x$.

`function s_aupdateby(a,x)` updates accumulator *a* by the value *x*. Call to this function replaces the assignment $a = a+x$. Both functions keep the minimum and the maximum values, the time integral and the time integral of squared assigned values.

`function[mean,min,max,variance,lastvalue] = s_accumstat(a)` returns the descriptive statistics on the accumulator *a*.

All statistical activities except assignment of accumulator values start after a user-defined warming up delay, for accumulators the user has to specify the initial values, mostly zeros.

3.3. Queues

Three usual types of queues (FIFO, LIFO, priority) with possibly limited capacity are implemented. Queues are represented by data structures with various fields used for statistics. Stored items are represented by the arrays of items structures, entry times, and priorities for priority queues. The following functions are available:

`function r = s_enqueue(q,i)` inserts the item *i* into the queue *q*. The output *r* specifies whether the insertion was successful (1) or not (0). Treatment of rejected arrivals is application dependent. Item data structure is specified by the user, the only compulsory field is *service* - the service duration when entering a queue. For all types of queues the item is placed at the end of an array.

`function [i,wt] = s_remove(q)` removes the next item from the queue *q*. The outputs are the item *i* and its waiting time *wt*. For priority queue the item is found by the Matlab function `min` in the array that contains the priorities. For all queues the item is physically removed by storing the empty values in the arrays.

`function s_nowait(q)` is used for statistics to record not waiting items in the queue *q*.

`function [...] = s_questat(q)` returns the statistics on queue *q*. The outputs are: mean queue length, mean waiting time, mean waiting of those who waited and left, maximum queue length, maximum waiting time, attempted arrival rate, effective arrival rate, rate of rejections, probability that the queue is full, number of attempted arrivals, number of rejected arrivals, number of not waiting arrivals and duration of statistics collection.

3.4. Model Function Structure

Simulation models are written as functions with a fixed structure. The input and output arguments are defined by the user. These are the parts of the model that have to be included in the following order:

- System functions
- User model initialization
- System model initialization
- User model functionality

The two system parts are the same for all models and they should not be modified. The two user parts can be any mixture of commands and local functions and of course any external functions can be called, typically functions for generation of random numbers. Anyway a very simple structure is suggested.

3.5. User Model Initialization

This part first tests the validity of model input arguments and initializes user model variables, if any. This optional code is application dependent. It is supposed to test the arguments of random number generators, array sizes, integrality, etc. Next some system variables have to be initialized by the user. This is in fact a part of the model specification. The following 8 system variables must be defined, the default initialization assumes a G/G/1 queuing model:

- Types of queues array. The items are 0/1/2 for FIFO, LIFO, and priority queues respectively.
- Maximum lengths of queues array. The items are non-negative integers or Inf for unlimited queues. Zeros for pure overflow models are accepted.
- Numbers of parallel channels array. It is assumed that each queue is served by several identical parallel channels. These three arrays must have the same length, but they can be empty.
- Data structure that represents entities (customers) stored in queues. In addition to already mentioned compulsory field *service*, there must be also the field *priority* if priority queues are part of the model. Other fields are user-defined, like for example attributes representing the history of the entity, types of entities, etc.
- Data structure that represents the user part of event notices. If stations are used (see later), the compulsory fields are *station* and *channel* used by the system event *end of service*.
- Warming-up delay for statistics collection.
- Number of tallies used in the model.
- Initial values of accumulators used in the model.

User code of the model is split into two parts because the values of the above system variables are needed for the system model initialization that prepares the sequencing set, the queues, and all statistics for the simulation run.

3.6. User Model Functionality

This part of user code follows the classical event-oriented paradigm. After scheduling at least one event, typically first arrival(s), breakdowns, etc., the simulation run is started by calling the function `s_imulation` followed by the simulation run evaluation, preferably implemented by another function. This function collects the statistics and assigns values to the model outputs.

3.7. Support for Queuing Systems

The tool is general; the only requirement is the possibility to express the model behavior in terms of events. Though the definition of the above 8 system variables has to be present, the values can be all empty, so there can be no queues in the models, no standard statistics, etc. Nevertheless typical application of discrete simulation is analysis and optimization of queuing systems, which is also our case. That's why we included a simple support that makes simulation of queuing networks simple and straightforward. We associate queues with a number of parallel channels serving the entities from the queue. This makes the so called *stations* supported by the following functions:

`function r = s_arrival(q,c)` is an arrival of the customer *c* to the station (queue) *q*. The result *r* specifies the outcome (0 = lost (rejected), 1 = enqueued, 2 = served without waiting). The user has to decide what to do in case of rejection due the limited capacity.

`function s_eos(ed)` is a system function activated by the engine `s_imulation` that is transparent to the user. It is an end of service specified by the data part *ed* of the corresponding event notice. For this purpose, if stations are used in the model, there are the two compulsory fields *station* and *channel* in the event notice data. After all necessary updates and statistics collection the following function is activated.

`function customer_leaving(s,c)` is the user's activity associated with the end of service to customer *c* in station *s*. Typically there is some decision about the next service, a call to `s_arrival`, or leaving the network.

`function r = s_stop(s,c)` stops the channel *c* in station *s*. The result *r* specifies the channel status (0 = idle, 1 = busy (the operation is completed), 2 = was already suspended).

`function r = s_resume(s,c)` re-activates the channel *c* in station *s*. The result *r* specifies the channel status (0 = idle, 1 = busy, 2 = suspended). Warning is given for the first two cases.

Additional statistics provided for stations by the function `s_questat` is the mean number of working channels and their utilization. Due to possible suspensions (failures) there is no simple relationship between these two figures.

The above mentioned functionality of stations is enabled by using the system events, so far only *end of service* was implemented. System events have negative numbers, are activated by the engine and for the user they are transparent. Also note that the functions

`s_arrival` and `customer_leaving` offer a sort of process-oriented view of the system dynamics. Call to `s_arrival` starts an internal process made of possible waiting in the queue and the service that ends when the customer appears as the argument in `customer_leaving`.

4. SIMULATION MODEL FOR EXAMPLES

The paper (Sklenar 2013) describes in detail how a model of a workshop made of two breaking down machines is built. For variance reduction examples we use a relatively simple model of a G/G/1 system given by the function:

```
function  [LQ,WQ,LQmax,WQmax,duration,L,W,
rho,lambda,mu] = GG1(intervals,services)
```

The inputs are vectors of intervals between arrivals and service durations of the arriving customers. The outputs are self-explaining; `duration` is the total time of collecting statistics. For this model the default initialization of the system variables is satisfactory, so the first part of user code just tests validity of the input arguments (skipped here), initializes the counter of arrivals and computes some outputs because the traffic rate must be tested to be less than 1:

```
na = length(intervals);
           % number of attempted arrivals
arr = 1;                  % arrival number
lambda=1/mean(intervals);  % arrival rate
mu = 1/mean(services);     % service rate
rho = lambda/mu;           % traffic rate
```

The model is in fact made of just one station, so the second part of user code is the following:

```
s_chedule(intervals(1),1,s_edata);
           % scheduling the first arrival
s_imulation;         % starting the engine
evaluation;        % experiment evaluation

function event(enumber,data,id)
  if enumber == 1               % arrival
     nextarrival;
  else
     error(['Unknown event']);
  end
end

function nextarrival   % customer arrival
  itm = s_item; % creating item structure
  itm.service = services(arr);
  arr = arr + 1;
  if arr<=na  % are there more arrivals ?
    s_chedule(s_time + intervals(arr),1,
      s_edata);  % schedule next arrival
  end
  s_arrival(1,itm);% arrival to station 1
end

function customer_leaving(qn,itm)
         % here this function is not used
end

function evaluation    % experiment eval.
  [LQ,WQ,...] = s_questat(1);
  W = WQ + mean(services);
  L = LQ + rho;
end
```

The function `evaluation` calls `s_questat` to get the first 5 output arguments and computes the remaining two outputs in obvious way. Note that the common event function `event` takes a trivial form; the function `customer_leaving` is not used by the model, but has to be present. The function `nextarrival` creates the entity structure, stores its service duration and schedules the next arrival (if any). The actual arrival is taken care of by the system function `s_arrival`.

5. VARIANCE REDUCTION EXAMPLES

The model described in the previous chapter can be used as a simulation "engine" in other demonstration models. So for example the following function runs repeatedly a given number of arrivals with intervals uniform in [*a*, *b*] and services uniform in [*c*, *d*]:

```
function [meanLQ,...] = ...
          sim_gg1(a,b,c,d,arrivals,runs)
da = b-a;
ds = d-c;
for i=1:runs
  display(['Run number: ' num2str(i)]);
  rands = rand(1,arrivals);
  ints = a + rands*da;       % intervals
  rands = rand(1,arrivals);
  servs = c + rands*ds;      % services
  [LQ(i)] = GG1(ints,servs);
end
meanLQ = mean(LQ);
...
```

From each run only the mean queue length is taken, at the end some descriptive statistics of the sample is computed and a histogram is shown. The variance reduction techniques dealt with in this paper are well known, see for example (Gentle 1998) or (Pidd 2004).

5.1. Antithetic Variables

This example shows how the use of antithetic variables decreases the sample variance. Here we want to find the mean queue length of the G/G/1 system with intervals uniform in [2,6] and services uniform in [1,5]. After taking a sample made of 500 independent runs, 100,000 arrivals each, we obtained the following results. The mean queue length *meanLQ* = 0.1538, the standard deviation *stdLQ* = 0.0023, so the coefficient of variation is *cvLQ* = 0.0148. As the measure of sample quality we are going to use the coefficient of variation (σ/μ) that gives the extent of variability independent of the actual value of the sample mean. We are dealing only with nonnegative values. It is well known that antithetic series can be created by replacing the underlying random numbers *u* by 1-*u* that are both uniformly distributed in (0,1). So we performed 250 pairs of runs, 100,000 arrivals each (same computation price), from each pair an average queue length was computed. The sample of 250 values thus obtained gave the same mean

queue length (at 4 decimal places) with *cvLQ* = 0.0083 which is a decrease by 44%. In our case we have a single queue system, so there is another way how to obtain two antithetic series by swapping random numbers used to generate intervals and service durations respectively. So after performing another 250 antithetic pairs of runs of the same length (again the same computation price) we again obtained the same mean queue length (at 4 places) with *cvLQ* = 0.0090 which is a decrease by 39%.

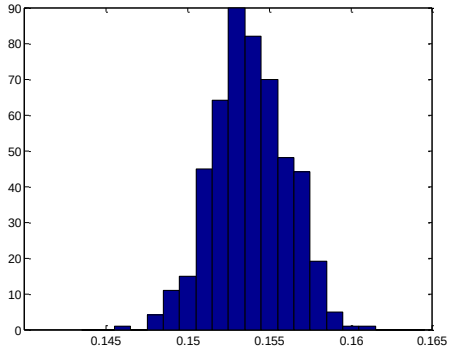

Figure 1: Sample histogram from 500 independent runs

Figures 1 and 2 show the histograms of the samples obtained by crude Monte Carlo and by using antithetic series (first case, the other one is very similar), both for the same 20 bins (0.144 : 0.001 : 0.164).

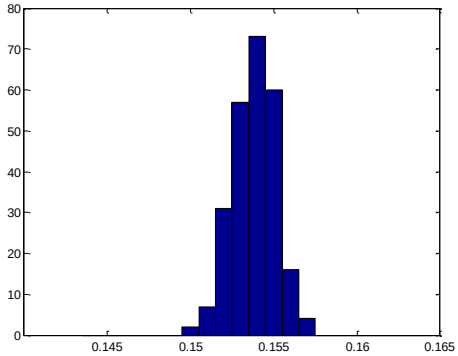

Figure 2: Sample histogram from 250 antithetic pairs of runs

Note that our very positive result is caused by the simplicity of the model. For more complicated models we would face the synchronization problem.

5.2. Common Random Numbers

This example shows how the use of common random numbers decreases the variance in situations where the purpose of the simulation is comparison of two systems. So let's compare the performance of the following two systems. The first is a G/G/1 system with intervals uniform in [5,15] and services uniform in [1,17] with traffic rate $\rho = \lambda/\mu = 0.9$. The other one is a G/M/1 system with same intervals uniform in [5,15] and exponentially distributed service with mean 8. Its traffic rate is $\rho = \lambda/\mu = 0.8$. As the comparison criterion we use the difference of the mean queue lengths $L_{Q2} - L_{Q1}$. After taking a sample made from 500 independent pairs of runs of the two systems (500 values of the difference) made of 100,000 arrivals each, we obtained the following results. The mean difference of the two queue lengths *difLQ* = 0.3047, the standard deviation *stddifLQ* = 0.0576, so the coefficient of variation is *cvdifLQ* = 0.1890. Note the very negative effect of the service variability. Though the traffic rate of the G/M/1 system is smaller, its mean queue length was always bigger, so the use of the coefficient of variation is still justified. In order to decrease the sample variance we performed another 500 pairs of runs of the same length, but in each pair the same underlying sequences of random numbers uniform in (0,1) were used. The mean difference of the two queue lengths was *difLQ* = 0.3068, the standard deviation *stddifLQ* = 0.0361, so the coefficient of variation is *cvdifLQ* = 0.1176 which is a decrease by 38%. Figures 3 and 4 show the histograms of the samples obtained by crude Monte Carlo and by using common random numbers, both for the same 24 bins (0 : 0.025 : 0.6).

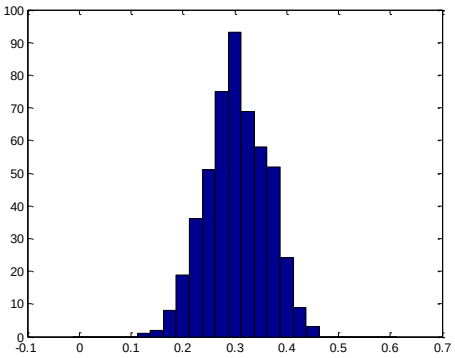

Figure 3: Sample histogram from 500 independent pairs of runs

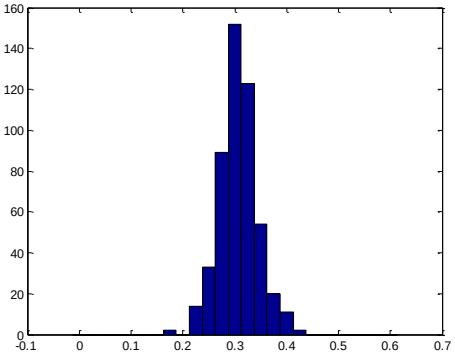

Figure 4: Sample histogram from 500 pairs of runs using common random numbers

The very positive result is again caused by the system simplicity, as synchronization in complex models is a complicated problem that may not be solved at all.

5.3. Control Variables

This example shows how the use of control variables decreases the sample variance. We want to find the mean queue length of the G/G/1 system with intervals uniform in [1,3] and services uniform in [1,2]. After taking a sample made of 500 independent runs, 50,000 arrivals each we obtained the following results. The mean queue length *meanLQ* = 0.0746, the standard deviation *stdLQ* = 0.0014, so the coefficient of variation is *cvLQ* = 0.0193. The technique of control variables is based on some knowledge about the internal working of the model. We first have to identify a variable such that its exact value is known and can be compared with the value obtained by simulation. Then we must know (at least approximately) the way how this value affects the simulation result, so we can perform a correction. In our example we select the control variable to be the mean service time T_S whose exact value is 1.5. The mean service time obtained by simulation is then $T_S + dT_S$. For simplicity we assume a linear relationship between the service time and the mean queue length, at least close to the actual values. So the change of the mean queue length caused by imperfect generation of service times is $c \times dT_S$. After simulation we perform the correction by subtracting this value from the mean queue length obtained by simulation. To find the value of the positive constant *c*, we may use a (simplified) analytical model or very long simulation runs. We performed 5 long simulation runs made of 10^6 arrivals each for the exact mean service 1.5 and another such 5 runs for the perturbed value 1.51 with services uniform in [1,2.02]. The difference of queue lengths (the two sample means) was then divided by 0.01 giving the value c = 0.455643. Using this constant we performed 500 corrected runs (50,000 arrivals each) with these results. The mean queue length *meanLQ* = 0.0744, the standard deviation *stdLQ* = 0.0012, so the coefficient of variation is *cvLQ* = 0.0171 which is a decrease by 11%.

Here the improvement is modest. Changing the perturbed value and increasing the numbers of long runs did not bring any considerable improvement. So we tried to use two control variables, the mean service time and the mean interval with the second one having obviously a negative effect on the queue length. So the correction subtracted from the simulated mean queue length was in this case $c_1 \times dT_S - c_2 \times dI_S$ where dI_S is the deviation between the exact mean interval 2 and the same value obtained by simulation. The value of c_1 is the same as the one used in the single control variable case, c_2 was obtained from two samples of long runs in similar way. Its value was c_2 = 0.248055. In this case the decrease of the coefficient of variation was 30%.

6. CONCLUSION

After using the Matlab based simulation tool to create various educational examples, see also two examples in (Sklenar 2013), we believe that the objective has been met. The user parts of the simulation models are very short, lucid, and all very similar. The differences are of cause given by the specific behavior of particular models. Due to the choice of the simple event-oriented paradigm, the code resembles very much programs in simulation languages of this type, for example Simscript II. Similar to these languages the user is relieved from "background" functionality like time control, queues management, statistics, etc. and can concentrate on the model behavior as such. All this is achieved without a need to learn a special-purpose simulation language. Intermediate Matlab programming skills are enough to create simulation models of medium size and complexity. As already mentioned, the whole model code is a single function with local functions, so though not recommended, it is possible to modify the system functions in any way. Having all the functionality under control and directly visible was in fact the main objective. Single function simulation models can thus be incorporated in programs for various repetitive experiments, optimization algorithms, etc. Of course there is a lot of room for further improvements. Both sequencing set and queues are implemented in a very simple inefficient way. Also security might become an issue; so far there are no restrictions at all. Though we believe that these drawbacks are not serious because the tool is not supposed to be applied in large simulation studies, its further development will address these issues.

REFERENCES

Bradley, A., 2007. *OptQuest for Arena - user's guide*. Rockwell Automation Technologies, Inc.

Deatcu, C., 2003. An object-oriented solution to ARGESIM comparison C6 - Emergency Department with MATLAB-DEVS2, *Simulation News Europe*, 38/39, 56.

Gentle, J.E., 1998. *Random Number Generation and Monte Carlo Methods*. Springer-Verlag.

Gray, M.A., 2007. Discrete event simulation: a review of SimEvents. *Computing in Science and Engineering*, 9(6), 62-66.

Kelton, W.D., Sadowski, R.P., Sadowski, D.A., 2006. *Simulation with Arena*. McGraw-Hill.

L'Ecuyer, P., 2007. Variance reduction greatest hits. *Proceedings of European Simulation and Modelling Conference ESM'2007*, 5-12. October 22-24, Malta.

Pidd, M., 2004. *Computer Simulation in Management Science.* 5th ed. John Wiley & Sons.

Sklenar, J., 2013. Tool for Discrete Event Simulation in MATLAB. *Proceedings of the 27th European Conference on Modelling and Simulation ECMS-2013*, 110-116, May 27-30, Alesund.

The MathWorks, Inc. 2005. *SimEvents user's guide*.

The MathWorks, Inc. 2005. *Simulink: a program for simulating dynamic systems, user guide*.