

MDESim - MATLAB Discrete Event Simulator

User's Guide

Jaroslav Sklenar

INTRODUCTION

One of the first decisions before starting building a simulation model is the nature of the simulation tool to be used. Or simply, will the simulation model be programmed (in a simulation language or a simulation library based on a general language), or will it be drawn by mouse by using a visual interactive tool? With respect to this fundamental decision we may identify two trends. 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, Extend, etc. is satisfactory and programming simulation models of such systems is often considered as a waste of time and money.

On the other hand there are 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. It is for example stochastic programming, finance, stochastic integration, reinforcement learning to mention just a few. Also systems with dynamically changing behavior and topology fall into this category. 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 are not supposed to be advanced programmers. In fact they 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.

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. MatlabDEV2 (Deatcu 2003) is a tool created primarily as a support for research and education of abstract Zigler's 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 clarity being the main objectives. The tool is not supposed to be used for large-scale computationally demanding simulation studies.

IMPLEMENTATION OF THE TOOL

These are the main facilities available in simulation languages and libraries:

- Time control (scheduling and canceling of future activities).
- Synchronization and communication of processes.
- Generation of random numbers from theoretical and empirical distributions.
- Automatic or user-friendly collection and evaluation of statistical data.
- Report generation.
- Control over the dynamically allocated memory.

- Facilities for user-friendly work with data structures used typically in discrete simulation models (queues).
- Diagnostics and facilities for observing model behavior.

The first and the most important decision is the time control paradigm. Though process oriented simulation has become a standard, we have decided to implement simpler “atomic” event oriented approach. It is much simpler to implement and the tool is not supposed to be used for large-scale simulation studies that would benefit from more natural and advanced process view of the world. So processes are represented by sequences of events with no synchronization and communication support.

Matlab has many functions for generation of random numbers, so only few functions were written for missing theoretical distributions (Erlang, triangular) and for empirical table distribution.

For statistics two typical objects (accumulator and tally) were implemented together with supporting functions computing elementary descriptive statistics.

For reporting there are Matlab functions for presentation graphics and advanced statistics. Also memory control was left up to the Matlab engine.

So far three basic types of queues were implemented – FIFO, LIFO, and priority queue.

No support for model debugging is available; again we assume that the tool will not be used for large-scale or complex models.

To simplify use of the tool as much as possible, all models are created as separate functions with most of the code same for all models. The user just adds code at given places, mostly just few lines. All is clearly explained by comments, so the best way of creating models is by modifying available typical examples. Next chapters provide more details.

Time Control

We assume that the reader is familiar with 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 *min* function 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. From the user’s point of view 4 functions and one system variable are available:

Current time is stored in the system variable `s_time` that of course should not be modified.

`function id = s_schedule(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_simulation` 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 functions. 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_simulation` that repeatedly removes the next event notice from the sequencing set, updates the time, and activates either the user function `event` or a hidden system event function. The run ends when the empty sequencing set is detected.

Statistics

With respect to time there are two types of statistics. Time dependent statistics (using Arena 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 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 needed to compute the variance and the number of updates.

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

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

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 needed to compute the variance. Both integrals are exact because they are made of sums of rectangular areas.

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

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 insertions is application dependent. Item data structure is specified by the user, the only compulsory field is *service* – the service duration when entering a queue. The implementation is very simple, 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 arrivals in the queue q .

function `[LQ,WQ,WQQ,LQmax,WQmax,lambda,l_eff,l_lost,pk,arrived,lost,nowaits,reff,util,duration] = s_questat(q)` returns the statistics on queue q . The outputs are: mean queue length, mean waiting time including no waits, 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, mean number of working channels (if used together with a number of parallel channels), utilization of channels and duration of statistics collection.

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 given order:

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

The two system parts are the same for all models and of course though not protected, 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.

User Model Initialization

This part first tests the validity of model input arguments and initializes user model variables, if any. This optional code is of course totally 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. Currently the following 8 system variables must be initialized:

1. Types of queues array. The items are 0/1/2 for FIFO, LIFO, and priority queues respectively.
2. Maximum lengths of queues array. The items are non-negative integers or Inf for unlimited queues. Zero for pure overflow queuing systems is accepted.
3. 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.
4. 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.
5. 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*.
6. Warming-up delay for statistics collection.
7. Number of tallies used in the model.
8. Initial values of accumulators used in the model.

There is the default initialization typical for models of G/G/1 queuing systems. So the user has to initialize only variables where other than default initial values are needed.

The 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.

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_simulation` followed by the run evaluation, preferably implemented by another function. This function collects the statistics and assigns values to the model outputs.

As already mentioned the simulation engine repeatedly activates the user function `event` that tests the event notice data and activates appropriate event functions that represent the model behavior. So lexicographically this part of code is made of few lines followed by user functions.

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_simulation` 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 current 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, they 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`.

EXAMPLE 1: G/G/c/K SYSTEM SIMULATION

We simulate a single queue system with *c* parallel identical channels and limited capacity $K \geq c$ (the maximum queue length is $K-c$). Intervals between arrivals of customers and service duration have any distribution, the only assumption is stability in queuing theory sense (arrival rate / c *service rate ≤ 1). The simulation model is implemented as a function:

```
function [LQ,WQ,WQQ,LQmax,WQmax,lambda,l_eff,l_lost,pk,arrived,lost,nowaits,reff,util,
         duration] = GGcK(intervals,services,c,K)
```

where the outputs are described above with the function `s_questat`, the first two arguments are vectors of the same length that contain intervals between arrivals and service durations of the particular customers. The other two arguments are system parameters. The model is thus general, the intervals and the service duration are generated externally. The function performs one simulation run, the number of customers is equal to the length of the two vectors.

User initialization part

The following is the full first part of the user code. It is made of self-explaining tests of input arguments and the initialization of two system variables. The other 6 are default or unused – see the next model.

```
% ***** Testing model arguments and *****
% ***** initialization of model "global" variables *****

na = length(intervals);      % number of attempted arrivals (customers)
arr = 1;                     % arrival number
if na~=length(services)
    error('Intervals and services don't have the same length');
end
if na==0
    error('At least one arrival required');
end
if min(intervals)<0
    error('There is a negative interval');
end
if min(services)<0
    error('There is a negative service duration');
end
if c<1
    error('Number of channels c < 1');
end
```

```

if c==Inf
    error('Number of channels c = Inf');
end
if K<c
    error('K < c');
end
if floor(c)~=c
    c = round(c);
    warning(['Number of channels not integer, rounded to ' num2str(c)]);
end
if floor(K)~=K
    K = round(K);
    warning(['System capacity not integer, rounded to ' num2str(K)]);
end

% ***** Initialization of system variables - user part *****
% ***** Assignment of other than default values - see above *****

s_maxq = [K-c];           % maximum lengths of queues array
s_channels = [c];        % numbers of channels array

```

User Model Behavior Part

The following commands schedule the first arrival. Then the engine is activated and the run is evaluated. Note that there is only one event, the arrival, implemented by the function `nextarrival`. It creates the customer entity, saves its service duration and if there are more arrivals to be scheduled it schedules the next one. Next call to `s_arrival` is the actual arrival to the station. The function `customer_leaving` is empty, but it has to be present. Evaluation just gets all the results about the station (queue).

```

% ***** Scheduling first event(s) *****

s_schedule(intervals(1),1,s_edata); % first arrival

% ***** Simulation run and evaluation *****

s_imulation;
evaluation;

% ***** User Functions *****
% ***** Functions "event" and "customer_leaving" must be included *****

function event(enumber,data,id) % common event routine
    % enumber = event number
    % data = user event data
    % id = event notice ID
    if enumber==1 % arrival
        nextarrival;
    else
        error(['Unknown event ' num2str(enumber)]);
    end
end %of event

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_schedule(s_time + intervals(arr),1,s_edata); % schedule next arrival
    end
    s_arrival(1,itm); % arrival to station 1
end %of nextarrival

function customer_leaving(qn,itm) % user defined reaction at departure from station
    % here the function is not used
end %of customer_leaving

```

```

function evaluation          % experiment evaluation = obtaining station statistics
    [LQ,WQ,WQQ,LQmax,WQmax,lambda,l_eff,l_lost,pk,arrived,lost,nowaits,reff,util,duration] =
    s_questat(1);
end %of evaluation

```

The following self-explaining function demonstrates the use of the model

```

function [LQ,pk,time] = GGcKEx(a,b,c,d,e,ch,K,arrivals,runs)
%
% intervals uniform in [a,b]
% service triangular (min c,likely d,max e)
% ch = number of channels
% K = system capacity
% arrivals = number of arrivals to be simulated
% runs = number of repetitions
%
% LQ = mean queue length
% pk = probability that the system is full
% time = CPU seconds taken
%
% Example use:
% [LQ,pk,time] = GGcKEx(1,2,1,4,8,2,6,10000,10)
%
time = cputime;
L = zeros(runs,1);
p = zeros(runs,1);
for i=1:runs
    display(['Run number: ' num2str(i)]);
    ints = uniform(a,b,rand(arrivals,1));           % intervals
    servs = triangular(c,d,e,rand(arrivals,1));     % service durations
    [L(i),WQ,WQQ,LQmax,WQmax,lambda,l_eff,l_lost,p(i)] = GGcK(ints,servs,ch,K);
end
LQ = mean(L);
pk = mean(p);
time = cputime - time;

```

The example use provides the following typical output:

```

>> [LQ,pk,time] = GGcKEx(1,2,1,4,8,2,6,10000,10)
Run number: 1
. . .
Run number: 10

LQ = 3.4685
pk = 0.3063
time = 28.7031

```

EXAMPLE 2: QUEUING NETWORK SIMULATION

Figure 1 depicts the abstraction of a workshop where machines are adjusted by two operations that may be repeated. The service times are triangular, the intervals are uniform, branching probabilities and parameters of the distributions are given in the figure. All times are given in minutes. We assume that the numbers of machines waiting for the two operations are not limited. Moreover the facilities for both operations may break down. The mean times between failures are exponentially distributed with means 180 and 120 respectively. After breakdown the facilities are repaired in uniformly distributed times in [5,25] and [7,20] respectively. We need a simple statistics on the number L of machines in the workshop and the time W a machine spends in the workshop (for both the mean, the minimum, the maximum, and the variance). Other usual performance parameters of the two stations will also be given.

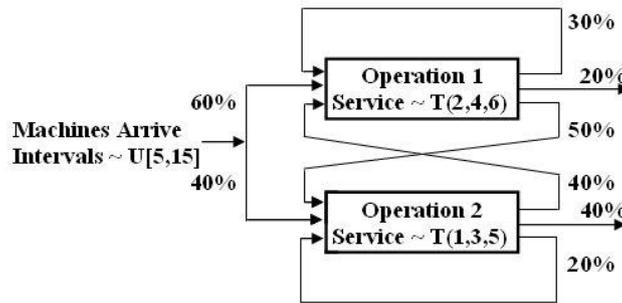


Figure 1. Workshop abstraction by a queuing network

The simulation model is implemented as the following function:

```
function [W, L, Wmin, Wmax, Wvar, Lmin, Lmax, Lvar] = exnet(duration,warming)
```

with self-explaining arguments and outputs. The function performs one simulation run and for simplicity the model parameters are directly included in the code. Next we give details on the two user parts of the model.

User initialization part

The following is the full first part of the user code. It is made of a simple test of the two arguments and initialization of system variables with other than default values:

```
% ***** Model dependent testing of model arguments and *****
% ***** initialization of model "global" variables (if any) *****

runduration = cputime;
if duration<=0
    error('Run duration not positive');
end
if duration<warming
    warning('Run duration < warming, no built-in statistics will be collected');
end

% ***** Initialization of system variables - user part *****
% ***** Assignment of other than default values - see above *****

s_types = [0 0];           % types of queues array (0/1/2 = FIFO/LIFO/Priority)
s_maxq = [Inf Inf];       % maximum lengths of queues array
s_channels = [1 1];       % numbers of channels array
s_item = struct('service',0,'sojourn',0); % customer entity (queue item) structure
% field 'service' is compulsory
% 'sojourn' = total time spent in network
s_warming = warming;      % warming-up delay for statistics collection
s_tallies = 1;            % number of tallies
s_accums = [0];          % initial values of accumulators
```

Referring to the text above we note that the model is made of two stations. Both queues are FIFO, unlimited, and served by single channels. Machines are represented by data structures that in addition to the compulsory *service* used to store the service time, contain the field *sojourn* where the total time spent in the workshop is computed. Warming up delay for statistics collection is given by the model argument; there is one tally for statistics on total network (*sojourn*) time and one accumulator initialized to zero for statistics on total number of machines in the workshop.

User Model Behavior Part

The following commands schedule the first arrival, the end of simulation run, and the first failures of the two stations. First arguments are the times of these events, the second arguments are their numbers. Then the engine is activated and the run is evaluated.

```

% ***** Scheduling first event(s) *****

s_schedule(5 + 10*rand,1,s_edata); % first arrival
s_schedule(duration,2,s_edata);   % end of run
s_schedule(exprnd(180),3,s_edata); % first breakdown of station 1
s_schedule(exprnd(120),4,s_edata); % first breakdown of station 2

% ***** Simulation run and evaluation *****

s_imulation;
evaluation;

```

Above commands are followed by user functions. The following one is activated by engine for each user event:

```

% ***** User Functions *****
% ***** Functions "event" and "customer_leaving" must be included *****

function event(enumber,data,id) % common event routine
% enumber = event number
% data = user event data
% id = event notice ID
switch enumber
    case 1 % arrival to network
        nextarrival;
    case 2 % end of run
        s_terminate;
    case 3 % breakdown of station 1
        s_stop(1,1);
        s_schedule(s_time + 5 + 25*rand,5,s_edata); % end of repair at 1 after U[5,25]
    case 4 % breakdown of station 2
        s_stop(2,1);
        s_schedule(s_time + 7 + 20*rand,6,s_edata); % end of repair at 2 after U[7,20]
    case 5 % end of repair at station 1
        s_resume(1,1);
        s_schedule(s_time + exprnd(180),3,s_edata); % next breakdown of station 1
    case 6 % end of repair at station 2
        s_resume(2,1);
        s_schedule(s_time + exprnd(120),4,s_edata); % next breakdown of station 2
    otherwise
        error(['Unknown event ' num2str(enumber)]);
end
end %of event

```

We note that all events are so simple that only arrival (event number 1) is written as a separate function. End of run (2) terminates the run by clearing the sequencing set. Breakdowns (3, 4) stop the operation of the particular channel and schedule the end of repair. Ends of repairs (5, 6) resume the operation of the channel and schedule the next breakdown. The next function is the activity associated with arrivals:

```

function nextarrival % customer arrival to network
s_aupdateby(1,1); % accumulator 1 = total number in network
s_schedule(s_time + 5 + 10*rand,1,s_edata); % next arrival after U[5,15]
itm = s_item; % machine created
if rand < 0.6
    itm.service = triangular(2,4,6,rand); % T[2,4,6]
    s_arrival(1,itm); % arrival to station 1
else
    itm.service = triangular(1,3,5,rand); % T[1,3,5]
    s_arrival(2,itm); % arrival to station 2
end
end %of nextarrival

```

The first command increments the accumulator that collects the statistics on the total number of machines in the network. Then after scheduling the next arrival, the item/machine data structure is created. After random branching, see Figure 1, the particular service time is generated and the actual arrival is done by calling the function `s_arrival` whose first argument is the station number. The next function is the activity associated with customers leaving the stations:

```

function customer_leaving(qn,itm) % user defined reaction at departure from station
% qn = station number
% itm = item structure (contains service time)
% itm.service changed to total time in station !
itm.sojourn = itm.sojourn + itm.service;
r = rand;
switch qn
case 1
if r<0.3 % branching after O1
itm.service = triangular(2,4,6,rand); % T[2,4,6]
s_arrival(1,itm); % arrival back to O1
elseif r<0.5
s_tupdate(1,itm.sojourn); % departure from network, tally 1 = sojourn time
s_aupdateby(1,-1); % accumulator 1 = total number in network
else
itm.service = triangular(1,3,5,rand); % T[1,3,5]
s_arrival(2,itm); % arrival to O2
end
case 2
if r<0.2 % branching after O2
itm.service = triangular(1,3,5,rand); % T[1,3,5]
s_arrival(2,itm); % arrival back to O2
elseif r<0.6
s_tupdate(1,itm.sojourn); % departure from network, tally 1 = sojourn time
s_aupdateby(1,-1); % accumulator 1 = total number in network
else
itm.service = triangular(2,4,6,rand); % T[2,4,6]
s_arrival(1,itm); % arrival to O1
end
otherwise
error(['Wrong station number ' num2str(qn) ' in customer_leaving']);
end
end %of customer_leaving

```

For both stations the sojourn time of the machine is updated and the branching decides whether the item proceeds to a station or leaves the network. If another service follows, the particular service time is generated followed by the arrival to the station. If the item leaves the network, the tally for sojourn time is updated by the total time spent in the network and the accumulator for the total number of items in the network is decremented. User run evaluation by the following function is very simple:

```

function evaluation % experiment evaluation
[LQ1,WQ1,WQQ1,LQmax1,WQmax1,lambda1,l_eff1,l_lost1,pk1,arrived1,lost1,nowaits1,reff1,
util1,duration1] = s_questat(1)
[LQ2,WQ2,WQQ2,LQmax2,WQmax2,lambda2,l_eff2,l_lost2,pk2,arrived2,lost2,nowaits2,reff2,
util2,duration2] = s_questat(2)
[W,Wmin,Wmax,Wvar,totalleft] = s_tallystat(1)
[L,Lmin,Lmax,Lvar,lastsize] = s_accumstat(1)
LL = LQ1 + LQ2 + reff1 + reff2 % L obtained otherwise
end %of evaluation

```

The first two commands get and display all available statistics on the two stations as explained above. The required model outputs are obtained from the tally and the accumulator. The last command is a check that computes the total number of items in the network from the two mean queue lengths and the two mean numbers of working channels that is the mean number of items being served.

The following is a formatted extract from outputs provided by a run of the length 100,000 minutes with warming up delay 100 minutes (and 'twister' generator initialized to default initial state). Note that the two ways of obtaining L provided equal values. The run took about 11s (MATLAB 7, E6700 3.2 GHz, 4GB RAM).

```

>>[W, L, Wmin, Wmax, Wvar, Lmin, Lmax, Lvar] = exnet(100000,100)

LQ1=1.2058   WQ1=6.9462   ...   util1=0.6938
LQ2=0.4831   WQ2=3.0470   ...   util2=0.4771

LL = 2.8597

```

```

W = 28.5642          Wmin = 1.0628
Wmax = 389.3541     Wvar = 1.1010e+003

L = 2.8597          Lmin = 0
Lmax = 17           Lvar = 4.6460
>>

```

EXAMPLE 3: OPTIMIZATION BY SIMULATION

Let's consider a hypothetical single queue single channel system with uniform distribution of inter-arrival intervals in $[0, x]$ and uniform distribution of service duration in $[0, y]$. The values x and y are under our control. Arrival rate can be decreased, the service rate can be increased with associated costs $c_A x$ and c_B / y respectively. Then we consider waiting cost $c_W L_Q$ where L_Q is the average length of the queue. The constants c_A and c_W are costs per appropriate time unit, c_B is the cost per unit service rate. We want to find such values of x and y that would minimize the total cost, so we solve the problem:

$$\min\{c_W L_Q(x, y) + c_A x + c_B / y \mid x > y, x > 0, y > 0\}$$

The first constraint guarantees stability (traffic rate $\rho < 1$). Moreover we assume an unconstrained ("not at a boundary") minimum because otherwise there would be no solution due to the open feasible set. The function $L_Q(x, y)$ can only be evaluated by simulation because we have a G/G/1 system with continuous distributions.

Creating a simulation model whose purpose is just obtaining the mean queue length is very easy. There is one station with one service channel, from user's point of view there is only one event type – customer arrival. The arrival function only schedules the next arrival and calls the system function `s_arrival` that moves the arriving customer to the station. The end of service is the system function, no user reaction is needed, so the function `customer_leaving` is left empty. For the code see EXAMPLE 1. The results provided by the model are the standard quantitative parameters of single queue systems provided by the system function `s_questat`, but here only the mean queue length over time is used.

For optimization a supporting function was written whose arguments are the parameters of the system being optimized together with the length of each simulation run expressed as the number of simulated arrivals and the number of repeated runs. The function returns the total cost for given optimization variables x and y . Figure 2 depicts the objective function that is the total cost for $c_W = 10$, $c_A = 1$, and $c_B = 30$. The second figure is a horizontal view showing the very flat minimum that is unfortunately typical for similar optimization problems. The mesh in Figure 2 was created with the step 0.05 for x in $[6, 8]$ and y in $[3, 5]$. So there are 1600 points, each of them was evaluated by 100 repetitions, each made of 20,000 simulated arrivals.

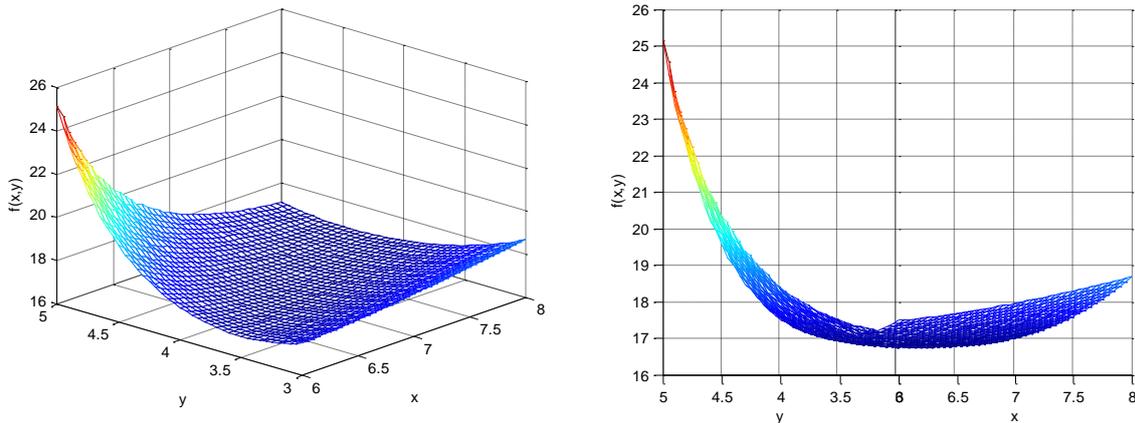


Figure 2. Objective function visualization

For optimization we used the standard Matlab function `fminsearch` that uses the robust moving simplex algorithm based only on function evaluations without making use of gradients. We believe that gradient evaluation by simulation would cause problems due to a very flat minimum and the well-known fact that at the optimum the gradient converges to zero. By the way the minimum was found at $x = 7.22$, $y = 4.10$ with total cost equal to 16.75 in our abstract currency units. Even for the very robust moving simplex algorithm it was necessary to decrease the argument and the objective

function tolerance to 0.01. This is due to the fact, that simulation based evaluation of the objective function is never accurate and so the objective function evaluation is strictly speaking inconsistent with the obvious optimization algorithm assumption that for given x and y the objective is known exactly. So the result would be a chaotic movement of small simplexes (triangles in our case) close to the minimum.

REFERENCES

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

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

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

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

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

Sklenar, J. "Discrete Simulation Language SIMSCRIPT T200". 1981. *Automatizace*, 4, 108-110.

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

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