Exercise 3 - Solutions

3.1

Building the model should pose no problem. Only, synchronizing the use of random numbers might require some attention. In order to control the state of the random number generator in Matlab, take a look at function *rng*

The state of the random number generator in Matlab can be controlled with the command

```
help RandStream;
```

The implementation of the model is as follows:

```
function
[y,n_in,n_out]=queue1(N,lambda,mu1,mu2,p,ua,ud1,ud2,up,antithetic)
%queueing model, exercise 3.1
%
%N            number of customers to simulate
%lambda       arrival rate
%mu1          service rate at server 1
%mu2          service rate at server 2
%p            customers leave system after server 1 with probability p,
%             otherwise leave the system
%ua           rng for arrivals
%ud1, ud2     rngs for service delays at servers 1 and 2
%up           rng for server selection
%antithetic 1 turns on antithetic variates, 0 turn them off
%
%
%All times exponentially distributed

t=0;          %Simulation clock

n_in=0;       %Counter
n_out=0;      %Counter

ta=0;         %Next arrival
td1=inf;      %Next departure at server 1
td2=inf;      %Next departure at server 2

s1=0;         %Customer service at server 1
s2=0;         %Customer service at server 2
nq1=0;        %Number in queue at server 1
nq2=0;        %Number in queue at server 2

tq=0;         %Total queueing time in the system


%First arrival
z=rand(ua);
if antithetic==1;z=1-z;end;
ta = -(1/lambda)*log(z);
```

```
%The main simulation loop
while n_out<N

    if ta<min(td1,td2)               %Next event is arrival

        n_in=n_in+1;
        tq=tq+nq1*(ta-t)+nq2*(ta-t);
        t=ta;

        if s1==0                     %Server is idle
            z=rand(ud1);
            if antithetic==1;z=1-z;end;
            td1=t-(1/mu1)*log(z);
            s1=1;
        else                         %Server is busy
            nq1=nq1+1;
        end

        z=rand(ua);                  %Next arrival
        if antithetic==1;z=1-z;end;
        ta=t-(1/lambda)*log(z);

    elseif td1<min(ta,td2)           %Next event is departure at server 1

        tq=tq+nq1*(td1-t)+nq2*(td1-t);
        t=td1;

        z=rand(up);
        if antithetic==1;z=1-z;end;

        if z>p                       %Customer enters server 2
            if s2==0                 %Server is idle
                z=rand(ud2);
                if antithetic==1;z=1-z;end;
                td2=t-(1/mu2)*log(z);
                s2=1;
            else                     %Server is busy
                nq2=nq2+1;
            end
        else                         %Customer leaves the system
            n_out=n_out+1;
        end


        if nq1>0                     %Waiting customers exist at server 1
            nq1=nq1-1;
            z=rand(ud1);
            if antithetic==1;z=1-z;end;
            td1=t-(1/mu1)*log(z);
            s1=1;
        else
            s1=0;
            td1=inf;
        end

    elseif td2<min(ta,td1)           %Next event is departure at server 2

        tq=tq+nq1*(td2-t)+nq2*(td2-t);
        t=td2;
        n_out=n_out+1;
        if nq2>0                        %Waiting customers exist at server 2
            nq2=nq2-1;
            z=rand(ud2);
            if antithetic==1;z=1-z;end;
            td2=t-(1/mu2)*log(z);
            s2=1;
        else
            s2=0;
            td2=inf;
        end

    end

end

y=tq/n_in;    %Return average total waiting time per customer
```

Now, to perform 10 independent replications with p=0.3, for instance, you could first create the random number streams with:

```
[ua,ud1,ud2,up]=RandStream.create('mrg32k3a','NumStreams',4);
```

and then write:

```
for i=1:10;y(i)=queue1(100,1,1/0.7,1/0.9,0.3,ua,ud1,ud2,up,0);end;
```

To use the same random numbers for p=0.8, you should first reset the random number streams:

```
reset(ua); reset(ud1); reset(ud2); reset(up);
```

and then simulate as above.

The results of 10 replications using independent and correlated sampling for configurations p=0.3 and p=0.8 could be as follows:

	Independent			CRN		
Replication	p=0.3	p=0.8	difference	p=0.3	p=0.8	difference
1	3.10	2.70	0.40	3.10	2.76	0.33
2	3.40	1.09	2.31	3.40	3.05	0.35
3	1.82	0.49	1.33	1.82	1.18	0.63
4	1.04	0.70	0.34	1.04	0.26	0.78
5	1.20	4.20	-3.00	1.20	0.37	0.83
6	9.07	2.02	7.04	9.07	1.08	7.99
7	3.37	0.72	2.64	3.37	1.04	2.32
8	1.13	0.59	0.53	1.13	0.98	0.14
9	1.08	1.01	0.06	1.08	0.78	0.29
10	4.85	1.05	3.80	4.85	1.08	3.77
average			1.54			1.74
variance			7.09			6.11

Variance reduction in this case is modest. Note, however, that this result is based on a rather limited sample size and is affected by the particular random number streams that were used in sampling. The results that you get may be different.

Results for antithetic variates are as follows:

	Independent			Antithetic		
Replication pair	1st	2nd	average	1st	2nd	average
1	2.76	1.16	1.96	2.76	0.97	1.87
2	3.05	1.24	2.14	3.05	1.23	2.14
3	1.18	0.95	1.07	1.18	0.66	0.92
4	0.26	0.84	0.55	0.26	1.80	1.03
5	0.37	1.14	0.75	0.37	1.33	0.85
average			1.30			1.36
variance			0.52			0.36

3.2

The Matlab-implementation of the simulation is as follows:

```
function c=machines(k,lambda,c_f,mu,c_r)
%Simulate availability of m machines
%lambda breakdown rate of all machines
%mu     repair rate of each machine
%k      repair capacity (number of repairmen
%c_f    cost per hour of an unavailable machine
%c_r    cost per hour of a repairman


T=800;                    %Total simulation time
m=5;                      %Number of machines

t=0;                      %Simulation clock
tb=exprnd(1/(m*lambda)); %The time of next breakdown
tr=inf;                   %The time of next repair completion

n_r=[];                   %Machines under repair
n_q=0;                    %Number queueing for repair

c=0;                      %Total cost


%The main simulation loop
while min(tb,tr)<T

    if tb<tr                      %Next event is breakdown

        %Update statistics
        c = c+(tb-t)*(c_f*(length(n_r)+n_q)+c_r*k);

        %Take to repairman or add in queue
        if length(n_r)<k
            n_r=sort([n_r tb+exprnd(1/mu)]);
        else
            n_q=n_q+1;
        end

        %Update simulation clock
        t = tb;

        %Next breakdown
        if length(n_r)+n_q<m
            tb=t+exprnd(1/(lambda*(m-length(n_r)-n_q)));
        else
            tb=inf;
        end

        %Next repair
        tr = n_r(1);

    else                            %Next event is repair completion

        %Update statistics
        c = c+(tr-t)*(c_f*(length(n_r)+n_q)+c_r*k);

        %Departure
        n_r=n_r(2:end);

        %Take next customer from repair queue, if queue is not empty
        if n_q>0
            n_r=sort([n_r tr+exprnd(1/mu)]);
            n_q = n_q-1;
        end

        %Update simulation clock
        t = tr;




        %Next service completion
        if not(isempty(n_r))
            tr = n_r(1);
        else
            tr=inf;
        end

        %Next breakdown (breakdown intensity increases as repair is
        %completed)
        tb=t+exprnd(1/(lambda*(m-length(n_r)-n_q)));

    end

end


%Update statistics for remaining time period
c = c+(T-t)*(c_f*(length(n_r)+n_q)+c_r*k);

%Average hourly cost
c=c/T;
```

In this exercise, we simulate all possible combinations of values for the following inputs (or *factors*) .

Factor	Level: -	Level: +
1*:* # of repairmen	2	4
2: Machine type	Time to failure: Expo(8) Cost of unavailability: 50$/hour	Time to failure: Expo(16) Cost of unavailability: 100$/hour
3: Repairman type	Time to repair: Expo(2) Cost of employment: 10$/hour	Time to repair: Expo(1.5) Cost of employment: 15$/hour

A full 2^3 factorial experiment can be represented with the matrix (x_{ij}), where rows represent factor level combinations and columns represent the factors. Element x_{ij} equals 1 if factor j is at its + -level in combination i. If factor j is at its - -level, x_{ij} equals -1.

Further, let r_{ik} denote the simulation response (average cost per hour of operating the machines) from the kth replication of factor combination i. The main effect of factor j corresponding to replication k is

$$e_j^{(k)} = \frac{1}{2^{3-1}} \sum_i x_{ij} r_{ik} \quad .$$

With n independent replications of the entire design, we get n observations of effects $e_j^{(k)}$. Calculating *t*-confidence intervals for the effects is straightforward.

Interaction effect of factors $j1$ and $j2$ is

$$e_{j_1 j_2}^{(k)} = \frac{1}{2^{3-1}} \sum_i x_{ij1} x_{ij2} r_{ik} \quad ,$$

and higher-order interactions are calculated analogously.

The resulting effects estimates (using a 0.05 level of confidence) are:

e_1	22.7 ± 3.2
e_2	3.0 ± 2.3
e_3	1.1 ± 4.3
e_{12}	-0.8 ± 3.2
e_{13}	6.4 ± 3.8
e_{23}	-0.5 ± 3.0

We see that none of the proposed changes would lead to decreased cost. Main effects of factors 1 and 2 are significant. The two-way interaction effect of 1 and 3 is significant as well.

3.3 (Demo)

Matlab-implementation of the simulation model:

```
function [tq,ts]=queue1(N,lambda,mu,p)
%Queueing model, exercise 6.1
%
%N            number of customers to simulate
%lambda       arrival rate
%mu           service rate at server 1
%p            customers re-enter server with probability 1-p,
%             otherwise leave the system
%rstate       state of random number generator
%
%All times exponentially distributed

t=0;          %Simulation clock

n_in=0;       %Counter
n_out=0;      %Counter

ta=0;         %Next arrival
td=inf;       %Next departure at server

s=[];         %Customer service at server, 0 first service, 1 second
service
si=[];        %Indices of customers at server
nq=[];        %Number in queue at server, 0 first service, 1 second service
nqi=[];       %Indices of customers at server

tq=zeros(N,1);         %queueing times for first N customers
ts=zeros(N,1);         %Service times for first N customers

%First arrival
ta = exprnd(1/lambda);

%The main simulation loop
while n_out<N

    if ta<td                        %Next event is arrival

        n_in=n_in+1;
        for i=1:length(nqi)
            if nqi(i)<=N
                tq(nqi(i))=tq(nqi(i))+(ta-t);
            end
        end
        t=ta;

        if isempty(s)               %Server is idle
            z=exprnd(1/mu);
            td=t+z;
            if n_in<=N
                ts(n_in)=ts(n_in)+z;
            end
            s=0;
            si=[n_in];
        else                        %Server is busy, enter queue
            nq=[nq 0];
            nqi=[nqi n_in];
        end

        ta=t+exprnd(1/lambda);

    elseif td<ta                    %Next event is departure at server

        for i=1:length(nqi)
            if nqi(i)<=N
                tq(nqi(i))=tq(nqi(i))+(td-t);
            end
        end
        t=td;

        z=rand;
        if z>p && s==0              %Customer re-enters server
            if isempty(nq)          %Queue is empty
                s=1;
                z=exprnd(1/mu);
                if si<=N
                    ts(si)=ts(si)+z;
                end
                td=t+z;
            else
                nq=[nq 1];          %Go to end of line
                nqi=[nqi si];

                s=nq(1);            %Take first customer from queue
                si=nqi(1);
                nq=nq(2:end);
                nqi=nqi(2:end);
                z=exprnd(1/mu);
                if si<=N
                    ts(si)=ts(si)+z;
                end
                td=t+z;
            end

        else

            if s<=N                 %We only count first N customers
                n_out=n_out+1;
            end

        if isempty(nq)
                s=[];
                si=[];
                td=inf;
            else
                s=nq(1);
                si=nqi(1);
                nq=nq(2:end);
                nqi=nqi(2:end);
                z=exprnd(1/mu);
                if si<=N
                    ts(si)=ts(si)+z;
```

```
                    end
                    td=t+z;
                end

            end
        end
    end

    tq=mean(tq);
    ts=mean(ts);
```

Simulation results of 10 independent replications are listed in the following table:

Replication	Average queuing time	Average service time
1	2.98	0.89
2	2.02	0.72
3	3.00	0.81
4	1.32	0.78
5	2.78	0.84
6	1.00	0.81
7	1.76	0.78
8	3.26	0.84
9	4.09	0.74
10	5.71	0.89
average	2.797	0.816

The average processing time across the replications is 0.816 minutes. This is somewhat less than the actual expected value (0.7+0.2*0.7=0.84) suggesting that we should adjust the uncontrolled estimate of 2.797 minutes of the expected average queueing time upwards. The controlled estimate now becomes

$$S_Y^2 = 0.003$$

$$\hat{C}_{XY} = 0.034$$

$$\hat{a}^* = 10.7$$

$$\bar{X}_C^* = 3.06$$

To have an idea, whether the controlled estimate actually is any better, the simulation was replicated 100 times. The resulting queueing time estimate was 3.12 minutes which suggests that the previously calculated controlled estimate (3.06 minutes) might be closer to the actual value of the performance measure than the uncontrolled one (2.797 minutes).

The results of the 100 replications were then used to calculate 10 estimates for the *queuing* time, each based on 10 replications. The results are as follows:

Estimate	Uncontrolled	Controlled
1	2.87	2.99
2	3.32	3.32
3	3.14	3.29
4	3.64	2.78
5	2.05	2.19
6	3.87	4.04
7	1.70	2.10
8	3.65	3.62
9	3.09	3.18
10	3.86	3.86
var	0.55	0.42

The variance of the 10 uncontrolled estimates is 0.55, whereas that for the controlled estimates is 0.42. Thus, we can conclude that the control variate procedure resulted in reduced variance for the estimate of the performance measure.