Solutions

4.1

We simply give the Matlab-code for the model.

```
function tout=queuenetwork(n,ent,seq,lambda,mu)

%Simulate a queuing network.
%
%length(lambda) different arrival processes
%length(mu) servers
%
%All times independent, exponential. Queue size infinite.
%
%n          simulate until n customers have left the system
%ent(i)     first server when entering the system from arrival process i
%seq(i)     destination server after leaving server i, length(mu)+1 to
leave the system
%lambda     inter-arrival time means of customers
%mu         service time means of servers

m=length(mu);        %Number of servers

t=0;                 %Simulation clock

ta=exprnd(lambda);   %Times of next customer arrivals
td=ones(1,m)*inf;    %Departure times of customers (set to infinite)

served=zeros(1,m);    %Customers being served
queue=cell(1,m);     %Queued customers

n_in=0;              %Number of customer arrivals
n_out=0;             %Number of customer departures

tin=[];              %Times of entering the system
tout=[];             %Total time in system


%The main simulation loop
while n_out<n

    if min(ta)<min(td)                    %Next event is arrival

        %Specify arrival process
        [z,i]=min(ta);

        %Update statistics
        n_in=n_in+1;
        tin=[tin;z];

        %Select server
        j=ent(i);

        %Add either to server or queue
        if served(j)==0
```

```
        served(j)=n_in;
        td(j)=z+exprnd(mu(j));
    else
        queue{j}=[queue{j} n_in];
    end

    %Update simulation clock
    t=z;

    %Next arrival
    ta(i)=z+exprnd(lambda(i));

else                            %Next event is departure

    %Get the index i of the server with the smallest departure time
    [z,i]=min(td);

    %Index of the customer leaving the server
    k=served(i);

    %Next server in sequence
    j=seq(i);

    %Customer either leaves the network or proceeds to next server
    if j>m                                   %Leave system
        n_out=n_out+1;
        tout=[tout; z-tin(k)];
    else                                     %Go to next server
        if served(j)==0
            served(j)=k;
            td(j)=z+exprnd(mu(j));
        else
            queue{j}=[queue{j} k];
        end
    end

    %Take next customer from queue at server i, if any exist
    if length(queue{i})>0
        served(i)=queue{i}(1);
        td(i)=z+exprnd(mu(i));
        queue{i}=queue{i}(2:end);
    else
        served(i)=0;
        td(i)=inf;
    end

    %Update simulation clock
    t=z;

end

%Command line output
str=['t: ' sprintf('%6.2f',t)];
for i=1:m
    if served(i)>0
        str=[str '   o|'];
    else
        str=[str '    |'];
    end
    str=[str repmat('o',1,length(queue{i})) repmat(' ',1,15-
        length(queue{i})) ];
end
```

```
    disp(str);
end
```

The following code illustrates the creation of the experimental design and the fitting of the metamodel.

```
%Simulation parameters
seq=[5 6 7 9 8 9 8 10 11 11 12];   %Sequence of servers in the simulation
lambda=[0.5 1 1 0.5];              %Mean inter-arrival times
mu=[0.4 0.7 0.5 0.3 0.3 0.8 0.2 0.5 0.4 0.5 0.2];   %Mean service times

%Construct a central composite desing (coded factor levels
x=ccdesign(11,'type','circumscribed','center',1);

%Form the design matrix that includes columns for the intercept as well
%as interaction and quadratic terms
X=[ones(size(x,1),1) x];
for i=1:10
    for j=i+1:11
        X=[X x(:,i).*x(:,j)];
    end
end
for i=1:11
    X=[X x(:,i).^2];
end

%Replicate the simulation to determine responses for each factor
%combination
r=5;                    %Replications per combination

for i=1:size(x,1)
    for k=1:r

        y(i,k)=queuenetwork(100,1:4,seq,lambda,(1+0.1*x(i,:)).*mu);

    end

    %disp(['comb:' num2str(i) '   mean resp:' num2str(Y(i))]);

end
Y=mean(y,2);          %Estimated mean responses for each factor combination
V=var(y,[],2);        %Estimated variances for each factor combination
C=eye(size(x,1)).*repmat(V,1,size(x,1)); %Corresponding covariance matrix

%Calculate the estimated weighted least squares estimates of the
%regression coefficients
beta=inv(X'*inv(C)*X)*X'*inv(C)*Y;

%Determine pseudo-values through jackknifing to determine confidence
%intervals for the regression coefficients
for i=1:r

    yi=y(:,[1:i-1 i+1:r]);
```

```
    Yi=mean(yi,2);
    Vi=var(y,[],2);
    Ci=eye(size(x,1)).*repmat(Vi,1,size(x,1));
    P(:,i)=inv(X'*inv(Ci)*X)*X'*inv(Ci)*Yi;

end
P=r*repmat(beta,1,r)-(r-1)*P;   %The pseudo-values are weighted
                                %combinations of the original coefficient
                                %estimates and the ones determined after
                                %deleting one replication

%Confidence intervals based on t-distribution
Pci=[mean(P,2)-tinv(0.975,r-1)*sqrt(var(P,[],2)/r)
mean(P,2)+tinv(0.975,r-1)*sqrt(var(P,[],2)/r)];

%Confidence intervals with ordinary least squares
s=regstats(Y,x,'quadratic');
```

4.2

We simply give the Matlab-codes that accomplish what was asked. The simulation of the queueing system:

```
function [d,ps]=GG1(n,lambda,mu,sigma,rstate)
%Simulate delays in queue for queuing model (GI/G/1)
%Arrivals exponential, service normally distributed.
%
%n           the number of customers
%lambda      the arrival rate of customers
%mu          mean service time of customers
%sigma       standard deviation of the service time
%rstate      state of random numbr generator

ua=rstate;          %Random stream for arrivals
ud=rstate+2*n;      %Random stream for service delays

t=0;                %Simulation clock
rand('state',ua);   %The time of next customer arrival
ta=exprnd(lambda);
td=inf;             %The time of next departure

ps=zeros(1,n);      %Waiting delay perturbations of customers

queue=[];           %Queued customers

n_in=0;             %Number of customer arrivals
n_out=0;            %Number of customer departures

d=zeros(1,n);       %Service delays


%The main simulation loop
while n_out<n

    if ta<td                       %Next event is arrival
```

```
        %Update statistics
        n_in = n_in+1;

        %Add either to server or queue
        if td==inf
            ud=ud+1;rand('state',ud);
            td = ta+normrnd(mu,sigma);
            d(n_in)= 0;
        else

            if isempty(queue)
                ps(n_in)=1;                %Propagate perturbation
            else
                ps(n_in)=1+ps(n_in-1);     %Propagate perturbation
            end

            queue=[queue ta];
        end

        %Update simulation clock
        t = ta;
        ua=ua+1;rand('state',ua);
        ta = ta+exprnd(1/lambda);

    elseif td<ta                           %Next event is departure

        %Update statistics
        n_out = n_out+1;

        %Update simulation clock
        t = td;

        %Take next customer from queue, if queue is not empty
        if not(isempty(queue))
            d(n_out+1)=td-queue(1);
            queue = queue(2:end);
            ud=ud+1;rand('state',ud);
            td = t+normrnd(mu,sigma);
        else
            td=inf;
        end

    end
end

d=mean(d);
ps=mean(ps);
```

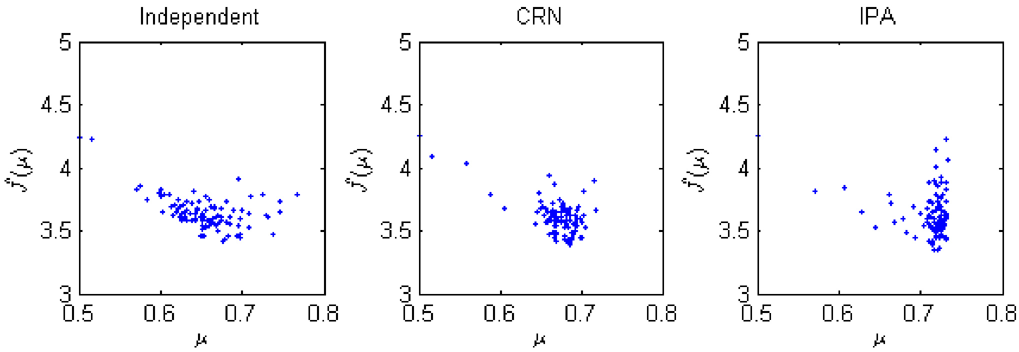
The optimization can be implemented as follows:

```
function [mu,Y,g,a]=sa(mu0,a0,M,R,mode,delta)
%Stochastic approximation to minimize the response of
%a GI/G/1 queueing model with respect to mean service time mu of
customers
%
%INPUT:
```

```
%
%mu0         Initial solution
%a0          Initial step size
%M           Maximum number of iterations
%R           Number of replications per simulation model evaluation
%mode        Type of gradient estimate
%            1: finite difference, independent replications
%            2: finite difference, common random numbers
%            3: infinitesimal perturbation
%delta       Perturbation of decision variable
%
%OUTPUT:
%
%mu          Trajectory of decision variable values
%y           Simulation model responses
%g           Gradient estimates
%a           step sizes

%Queueing model parameters
n=100;       %Number of customers to simulate
lambda=1;    %Arrival rate of customers
sigma=0.1;   %Standard deviation of service time
c=2;         %Cost

mu=mu0;
k=0;         %iteration count

while k<M

    k=k+1;

    %Optimization criterion at current solution
    for i=1:R
        [D(i) p(i)]=GG1(100,1,mu(k),0.1,(i-1)*500);
    end
    Y(k)=mean(D)+c/mu(k);

    %Gradient estimate
    if mode==1
        for i=1:R
            d(i)=GG1(100,1,mu(k)+delta,0.1,(M+k-1)*R*500+(i-1)*500);
        end
        y(k)=mean(d)+c/(mu(k)+delta);
        g(k)=(y(k)-Y(k))/delta;
    elseif mode==2
        for i=1:R
            d(i)=GG1(100,1,mu(k)+delta,0.1,(k-1)*R*500+(i-1)*500);
        end
        y(k)=mean(d)+c/(mu(k)+delta);
        g(k)=(y(k)-Y(k))/delta;
    elseif mode==3
        y(k)=0;
        g(k)=mean(p)-c/(mu(k)^2);
    end

    %Step size update
    a(k)=a0/sqrt(k);

    %Take a step in the direction of the gradient
    mu(k+1)=max(0,mu(k)-g(k)*a(k));

    %Command line output
```

```
    disp(['k: ' num2str(k) '    mu: ' num2str(mu(k)) '    Y: '
num2str(Y(k)) '    y: ' num2str(y(k)) '    g: ' num2str(g(k)) '    a: '
num2str(a(k)) ]);

end
```

Try, for instance, using μ^0=0.5, a_0=0.01, δ=0.01. Do at least 100 iterations. Here, two different numbers of replications are tried. Firstly, making only 2 replications gives us the following result. Both IPA and common random numbers seemingly produce better gradient estimates compared finite difference estimation with independent sampling (we see this, since the points in the corresponding figures are much more concentrated).



Now, making 5 replications per objective function evaluation leads to following result. Obviously, both objective function and gradient estimates are now more accurate.