ECS 256 - Problem set 1

Olga Prilepova, Christopher Patton, Alexander Rumbaugh, John Chen, Thomas Provan

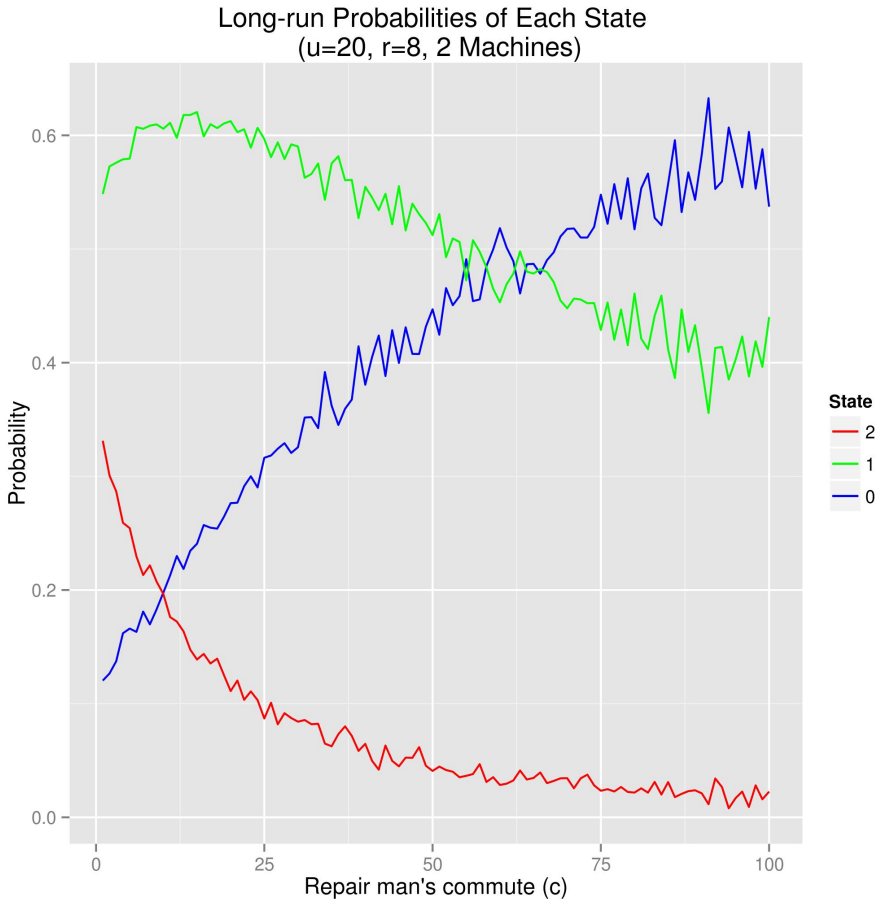
For this assignment, we adopt the following notation. Suppose W is a random variable exponentially distrubted with mean μ. Then $W \sim \mathcal{E}(\frac{1}{\mu})$.

Problem 1

(a) $w = 0 \cdot \pi_0 + 1 \cdot \pi_1 + 2 \cdot \pi_2 = \pi_1 + 2\pi_2$.

(b) Program is `1B.R`.

(c) Program is `1C.R`.



Problem 2

(a) Let $D \sim \mathcal{E}(1/d)$ be a random variable corresponding to the call druation and $R \sim \mathcal{E}(1/r)$ the time between queued calls. The balance equations for the call system:

$$\pi_{(i,j)}\lambda_{(i,j)} = \pi_{(i,j-1)}\lambda_{(i,j-1)}p_{(i,j-1)(i,j)} + \pi_{(i,j+1)}\lambda_{(i,j+1)}p_{(i,j+1)(i,j)} \quad (\forall \quad 0 < j < q, 0 < i < n)$$

$$\pi_{(i,0)}\lambda_{(i,0)} = \pi_{(i+1,0)}\lambda_{(i+1,0)}p_{(i+1,0)(i,0)} + \pi_{(i,1)}\lambda_{(i,1)}p_{(i,1)(i,0)} \quad (\forall \quad i < n)$$

$$\pi_{(i,q)}\lambda_{(i,q)} = \pi_{(i,q-1)}\lambda_{(i,q-1)}p_{(i,q-1)(i,q)} \quad (\forall \quad i > 1)$$

$$\pi_{(1,q-1)}\lambda_{(i,q-1)} = \pi_{(i-1,q)}\lambda_{(i-1,q)}p_{(i-1,q)(i,q-1)} \quad (\forall \quad i > 1)$$

$$\pi_{(n,0)}\lambda_{(n,0)} = \pi_{(n,1)}\lambda_{(n,1)}p_{(n,1)(n,0)}$$

$$\pi_{(1,q)}\lambda_{(1,q)} = \pi_{(1,q-1)}\lambda_{(1,q-1)}p_{(1,q-1)(1,q)}$$

$$\pi_{(1,0)} + \cdots + \pi_{(1,q)} + \cdots + \pi_{(n,0)} + ... + \pi_{(n,q)} = 1$$

Proportion of rejected calls. The long-run probability that the call center is in a state in which it's rejecting calls is the following quantity:

$$\sum_{i=1}^{n} \pi_{(i,q)}$$

Proportion of nurse idle time. To qunatify this value, we introduce a new state *idle*. The only possible transition to this state is from $(1, 0)$. By (8.5) in the book, the proportion of time spent in this state is:

$$\pi_{idle} = \frac{\pi_{(1,0)}\lambda_{(1,0)}p_{(1,0),idle}}{\lambda_{idle}}$$

The time spent in state $(1, 0)$ is the random variable $Z = \min(R, D)$, so $p_{(1,0),idle} = P(Z = D) = \frac{1/d}{1/r+1/d}$. Finally, $\lambda_{idle} = 1/r$.

Mean time spent in state $(1, 1)$. The possible transitions from this state are $(1, 0)$ (nurse finishes call, takes next on queue) and $(1, 2)$ (call comes in before nurse finishes

with current call). The time spent in this state is then $Z_{(1,1)} = \min\{D, R\}$. By theorem, $Z_{(1,1)} \sim \mathcal{E}(1/d + 1/r)$. Thus, the mean time spent in this state is:

$$\frac{1}{\frac{1}{d} + \frac{1}{r}}$$

Given the queue is emtpy, long-run probability that A_t is greater than 1. The sum of the stationary probabilities where $j = 0$ and $i > 1$:

$$\sum_{i=2}^{n} \pi_{(i,0)}$$

Appendix

Problem 1(b)

```
1
2 # DES.R:  R routines for discrete-event simulation (DES), with an example
3
4 # matrix version; data frame allows character event types, but much too slow
5
6 # all data is stored in an R environment variable that will be referrred
7 # to as simlist below
8
9 # the simlist will consist of the following components:
10 #
11 #       currtime:  current simulated time
12 #       evnts:  the events list, a matrix
13 #       reactevent:  event handler, user-supplied; creates new
14 #                    events upon the occurrence of an old one;
15 #                    e.g. job arrival triggers either start of
16 #                    service for the job or queuing it; call form is
17 #                    reactevent(evnt,simlist)
18 #       dbg:  if TRUE, will print evnts above after each event
19 #             scheduling action, and enter R browser for single-stepping
20 #             etc.
21
22 # the application code can add further application-specific data to
23 # simlist, e.g. total job queuing time
24
```

```
25 # each event will be represented by a matrix row consisting of:
26 #
27 #    occurrence time
28 #    event type (user-defined numeric code)
29 #
30 # and application-specific information, if any
31
32 # library functions (do not alter):
33 #
34 #       newsim:  create a new simlist
35 #       insevnt:  insert a new event into evnts in the simlist
36 #       schedevnt:  schedule a new event (determine its occurrence time
37 #                   and call insevnt())
38 #       getnextevnt:  pulls the earliest event from the event list,
39 #                     process it, and update the current simulated
40 #                     time
41 #       mainloop:  as the name implies
42 #       appendtofcfsqueue:  append job to a FCFS queue
43 #       delfcfsqueue:  delete head of a FCFS queue
44
45 # outline of a typical application:
46
47 #    mysim <- newsim()    create the simlist
48 #    set reactevent in mysim
49 #    set application-specific variables in mysim, if any
50 #    set the first event in mysim$evnts
51 #    mainloop(mysim,mysimtimelim)
52 #    print results
53
54 # create a simlist, which will be the return value, an R environment
55 newsim <- function(dbg=F) {
56    simlist <- new.env()
57    simlist$currtime <- 0.0  # current simulated time
58    simlist$evnts <- NULL  # event list
59    simlist$dbg <- dbg
60    simlist
61 }
62
63 # insert event evnt into evnts in simlist
64 insevnt <- function(evnt,simlist) {
65    # if the event list is empty, set it to consist of evnt and return
```

```
66      if (is.null(simlist$evnts)) {
67          simlist$evnts <- matrix(evnt,nrow=1)
68          return()
69      }
70      # otherwise, find insertion point
71      inspt <- binsearch(simlist$evnts[,1],evnt[1])
72      # now "insert," by reconstructing the matrix; we find what portion of
73      # the current matrix should come before evnt and what portion should
74      # come after it, then string everything together
75      before <- if (inspt == 1) NULL else simlist$evnts[1:(inspt-1),]
76      nr <- nrow(simlist$evnts)
77      after <- if (inspt <= nr) simlist$evnts[inspt:nr,] else NULL
78      simlist$evnts <- rbind(before,evnt,after)
79      rownames(simlist$evnts) <- NULL
80  }
81
82  # schedule new event in evnts in simlist; evnttime is the time at
83  # which the event is to occur; evnttype is the event type; appdata is
84  # a vector of numerical application-specific data
85  schedevnt <- function(evnttime,evnttype,simlist,appdata=NULL) {
86      evnt <- c(evnttime,evnttype,appdata)
87      insevnt(evnt,simlist)
88  }
89
90  # start to process next event (second half done by application
91  # programmer via call to reactevnt() from mainloop())
92  getnextevnt <- function(simlist) {
93      head <- simlist$evnts[1,]
94      # delete head
95      if (nrow(simlist$evnts) == 1) simlist$evnts <- NULL else
96          simlist$evnts <- simlist$evnts[-1,,drop=F]
97      return(head)
98  }
99
100 # main loop of the simulation
101 mainloop <- function(simlist,simtimelim) {
102     while(simlist$currtime < simtimelim) {
103         head <- getnextevnt(simlist)
104         # update current simulated time
105         simlist$currtime <- head[1]
106         # process this event (programmer-supplied ftn)
```

```
107         simlist$reactevent(head,simlist)
108         if (simlist$dbg) {
109             print("event␣occurred:")
110             print(head)
111             print("events␣list␣now")
112             print(simlist$evnts)
113             browser()
114         }
115     }
116 }
117
118 # binary search of insertion point of y in the sorted vector x; returns
119 # the position in x before which y should be inserted, with the value
120 # length(x)+1 if y is larger than x[length(x)]; this could be replaced
121 # by faster C code
122 binsearch <- function(x,y) {
123     n <- length(x)
124     lo <- 1
125     hi <- n
126     while(lo+1 < hi) {
127         mid <- floor((lo+hi)/2)
128         if (y == x[mid]) return(mid)
129         if (y < x[mid]) hi <- mid else lo <- mid
130     }
131     if (y <= x[lo]) return(lo)
132     if (y < x[hi]) return(hi)
133     return(hi+1)
134 }
135
136 # appendtofcfsqueue() and delfcfsqueuehead() below assume the
137 # application code has one or more queues, each queue stored as a
138 # list-of-lists, with each individual list being the information for one
139 # queued job; note that one must initialize the list-of-lists as NULL
140
141 # appends jobtoqueue to the given queue, assumed of the above form;
142 # the new, longer list is returned
143 appendtofcfsqueue <- function(queue,jobtoqueue) {
144     lng <- length(queue)
145     queue[[lng+1]] <- jobtoqueue
146     queue
147 }
```

```
148
149 # deletes head of queue; assumes list-of-lists structure as decribed
150 # above; returns the head and new queue
151 delfcfsqueuehead <- function(queue) {
152     qhead <- queue[[1]]
153     newqueue <- queue[-1]
154     # careful!--an empty list is not NULL
155     if (length(queue) == 1) newqueue <- NULL
156     list(qhead=qhead,newqueue=newqueue)
157 }
158
159
160
161
162 ## Start of our code.
163
164
165
166 mr1b <- function(u, r, k, timelim, dbg=F) {
167   # Event types:
168   # 1 - machine failed
169   # 2 - machine repaired
170   # An event is a tuple c(time, event_type).
171
172   # Set up simulation list, specify event handler.
173   simlist <- newsim(dbg)
174   simlist$reactevent <- mr1breact
175
176   # Parameters required by mr1breact()
177   simlist$lambda_u = 1/u
178   simlist$lambda_r = 1/r
179   simlist$k <- k # total machines
180   simlist$i <- k # up machines (initial state)
181   simlist$time <- rep(0, k)
182
183   # We must generate the first event and handle it.
184   # Since the simulation starts with all of the
185   # machines running, the first event will be a
186   # failure.
187   ttf <- min(rexp(k, 1/u))
188   simlist$totaltime <- ttf # Another parameter: maintain the duration
```

```
189                                 # of the simulation. Used to calculate the
190                                 # average number of machines running.
191     simlist$time[k] <- ttf
192     schedevnt(ttf, 1, simlist)
193
194     # Enter main loop (calls mr1breact()).
195     mainloop(simlist, timelim)
196
197     # Report average number of machines running.
198     simlist$time <- simlist$time / simlist$totaltime
199     w <- 0
200     for (i in 1 : k)
201     {
202       w <- w + (simlist$time[i] * i)
203     }
204     print("Average number of machines running")
205     print(w)
206 }
207
208 # Our reactevent(). Transition to new state
209 # and generate next event.
210 mr1breact <- function(evnt, simlist) {
211     etype <- evnt[2]
212
213     # Transition state.
214
215     if (etype == 1) # failure
216     {
217       simlist$i = simlist$i - 1
218     }
219
220     else if (etype == 2) # repair
221     {
222       simlist$i = simlist$i + 1
223     }
224
225     # Choose next event.
226
227     if (simlist$i == 0)
228     {
229       tte <- min(rexp(simlist$k, simlist$lambda_r))
```

```
230     etype <- 2
231   }
232
233   else if (simlist$i == simlist$k)
234   {
235     tte <- min(rexp(simlist$k, simlist$lambda_u))
236     etype <- 1
237     simlist$time[simlist$i] <- simlist$time[simlist$i] + tte
238   }
239
240   else
241   {
242     ttf <- min(rexp(simlist$i, simlist$lambda_u))
243     ttr <- min(rexp(simlist$k - simlist$i, simlist$lambda_r))
244     if (ttf < ttr)
245     {
246       tte <- ttf
247       etype <- 1
248     }
249     else
250     {
251       tte <- ttr
252       etype <- 2
253     }
254     simlist$time[simlist$i] <- simlist$time[simlist$i] + tte
255   }
256
257   schedevnt(simlist$currtime + tte, etype, simlist)
258   simlist$totaltime <- simlist$totaltime + tte
259
260   #print("----------")
261   #print(simlist$i)
262   #print("next event")
263   #print(etype)
264   #print(simlist$currtime)
265
266 }
```

Problem 1(c)

```
1
2 # DES.R:  R routines for discrete-event simulation (DES), with an example
```

```
3
4 # matrix version; data frame allows character event types, but much too slow
5
6 # all data is stored in an R environment variable that will be referrred
7 # to as simlist below
8
9 # the simlist will consist of the following components:
10 #
11 #       currtime:  current simulated time
12 #       evnts:  the events list, a matrix
13 #       reactevent:  event handler, user-supplied; creates new
14 #                    events upon the occurrence of an old one;
15 #                    e.g. job arrival triggers either start of
16 #                    service for the job or queuing it; call form is
17 #                    reactevent(evnt,simlist)
18 #       dbg:  if TRUE, will print evnts above after each event
19 #             scheduling action, and enter R browser for single-stepping
20 #             etc.
21
22 # the application code can add further application-specific data to
23 # simlist, e.g. total job queuing time
24
25 # each event will be represented by a matrix row consisting of:
26 #
27 #    occurrence time
28 #    event type (user-defined numeric code)
29 #
30 # and application-specific information, if any
31
32 # library functions (do not alter):
33 #
34 #       newsim:  create a new simlist
35 #       insevnt:  insert a new event into evnts in the simlist
36 #       schedevnt:  schedule a new event (determine its occurrence time
37 #                   and call insevnt())
38 #       getnextevnt:  pulls the earliest event from the event list,
39 #                     process it, and update the current simulated
40 #                     time
41 #       mainloop:  as the name implies
42 #       appendtofcfsqueue:  append job to a FCFS queue
43 #       delfcfsqueue:  delete head of a FCFS queue
```

```
44
45 # outline of a typical application:
46
47 #     mysim <- newsim()     create the simlist
48 #     set reactevent in mysim
49 #     set application-specific variables in mysim, if any
50 #     set the first event in mysim$evnts
51 #     mainloop(mysim,mysimtimelim)
52 #     print results
53
54 # create a simlist, which will be the return value, an R environment
55 newsim <- function(dbg=F) {
56     simlist <- new.env()
57     simlist$currtime <- 0.0  # current simulated time
58     simlist$evnts <- NULL  # event list
59     simlist$dbg <- dbg
60     simlist
61 }
62
63 # insert event evnt into evnts in simlist
64 insevnt <- function(evnt,simlist) {
65     # if the event list is empty, set it to consist of evnt and return
66     if (is.null(simlist$evnts)) {
67         simlist$evnts <- matrix(evnt,nrow=1)
68         return()
69     }
70     # otherwise, find insertion point
71     inspt <- binsearch(simlist$evnts[,1],evnt[1])
72     # now "insert," by reconstructing the matrix; we find what portion of
73     # the current matrix should come before evnt and what portion should
74     # come after it, then string everything together
75     before <- if (inspt == 1) NULL else simlist$evnts[1:(inspt-1),]
76     nr <- nrow(simlist$evnts)
77     after <- if (inspt <= nr) simlist$evnts[inspt:nr,] else NULL
78     simlist$evnts <- rbind(before,evnt,after)
79     rownames(simlist$evnts) <- NULL
80 }
81
82 # schedule new event in evnts in simlist; evnttime is the time at
83 # which the event is to occur; evnttype is the event type; appdata is
84 # a vector of numerical application-specific data
```

```
85 schedevnt <- function(evnttime,evnttype,simlist,appdata=NULL) {
86    evnt <- c(evnttime,evnttype,appdata)
87    insevnt(evnt,simlist)
88 }
89
90 # start to process next event (second half done by application
91 # programmer via call to reactevnt() from mainloop())
92 getnextevnt <- function(simlist) {
93    head <- simlist$evnts[1,]
94    # delete head
95    if (nrow(simlist$evnts) == 1) simlist$evnts <- NULL else
96       simlist$evnts <- simlist$evnts[-1,,drop=F]
97    return(head)
98 }
99
100 # main loop of the simulation
101 mainloop <- function(simlist,simtimelim) {
102    while(simlist$currtime < simtimelim) {
103       head <- getnextevnt(simlist)
104       # update current simulated time
105       simlist$currtime <- head[1]
106       # process this event (programmer-supplied ftn)
107       simlist$reactevent(head,simlist)
108       if (simlist$dbg) {
109          print("event␣occurred:")
110          print(head)
111          print("events␣list␣now")
112          print(simlist$evnts)
113          browser()
114       }
115    }
116 }
117
118 # binary search of insertion point of y in the sorted vector x; returns
119 # the position in x before which y should be inserted, with the value
120 # length(x)+1 if y is larger than x[length(x)]; this could be replaced
121 # by faster C code
122 binsearch <- function(x,y) {
123    n <- length(x)
124    lo <- 1
125    hi <- n
```

```
126     while(lo+1 < hi) {
127        mid <- floor((lo+hi)/2)
128        if (y == x[mid]) return(mid)
129        if (y < x[mid]) hi <- mid else lo <- mid
130     }
131     if (y <= x[lo]) return(lo)
132     if (y < x[hi]) return(hi)
133     return(hi+1)
134 }
135 
136 # appendtofcfsqueue() and delfcfsqueuehead() below assume the
137 # application code has one or more queues, each queue stored as a
138 # list-of-lists, with each individual list being the information for one
139 # queued job; note that one must initialize the list-of-lists as NULL
140 
141 # appends jobtoqueue to the given queue, assumed of the above form;
142 # the new, longer list is returned
143 appendtofcfsqueue <- function(queue,jobtoqueue) {
144     lng <- length(queue)
145     queue[[lng+1]] <- jobtoqueue
146     queue
147 }
148 
149 # deletes head of queue; assumes list-of-lists structure as decribed
150 # above; returns the head and new queue
151 delfcfsqueuehead <- function(queue) {
152     qhead <- queue[[1]]
153     newqueue <- queue[-1]
154     # careful!--an empty list is not NULL
155     if (length(queue) == 1) newqueue <- NULL
156     list(qhead=qhead,newqueue=newqueue)
157 }
158 
159 
160 
161 
162 ## Start of our code.
163 
164 library(ggplot2)
165 
166 # Plot long run probabilties of each state.
```

```
plotc <- function(c){
  u <- 20
  r <- 8
  timelim <- 10000
  c_column <- rep(NA, length(c))
  p0_column <- rep(NA, length(c))
  p1_column <- rep(NA, length(c))
  p2_column <- rep(NA, length(c))
  for (c_i in c){
    result_vector = mr1c(u, r, c_i, timelim)
    c_column[c_i] = c_i
    p0_column[c_i] = result_vector[1]
    p1_column[c_i] = result_vector[2]
    p2_column[c_i] = result_vector[3]
  }
  X <- data.frame(c_column, p0_column, p1_column, p2_column)

  ggplot(X) + geom_line(aes(y=p0_column, x=c_column, color="red")) +
              geom_line(aes(y=p1_column, x=c_column, color="green")) +
              geom_line(aes(y=p2_column, x=c_column, color="blue")) +
              xlab("Repair man's commute (c)") + ylab("Probability") +
              scale_colour_manual(name = "State",
                                  labels = c("2", "1", "0"),
                                  values = c("red", "green", "blue")) +
              ggtitle("Long-run Probabilities of Each State \n (u=20, r=8, 2 Mac

}



#1.c
#event: curtime eventtype(1=fail,2=repair) machine_num(1,2) timeUp timeDown
#mr1c: status(1=offsite,2=onsite) time_when_will_be_available

mr1c <- function(u, r, c, timelim, dbg=F) {
  simlist <- newsim(dbg)
  simlist$reactevent <- mr1creact
  simlist$lambda_u = 1/u
  simlist$lambda_r = 1/r
  simlist$c <- c #time it take for the mr1c to get to the machines
```

```
208   mr1c <- c(1, 0) #offsite, available since time 0 [only matters for onsite]
209   simlist$mr1c <- mr1c
210
211   #start with both machines running. find time when each one will fail
212   ttf1 <- rexp(1, simlist$lambda_u)
213   ttf2 <- rexp(1, simlist$lambda_u)
214
215   #schedule them (scheduer will sort them by whichever occurs first)
216   schedevnt(ttf1, 1, simlist, c(1)) #c(1st machine)
217   schedevnt(ttf2, 1, simlist, c(2))
218
219   simlist$results = c(0,0,0) # times for 0 machines simlist$results[1], 1 machi
220   simlist$totaltime = 0
221   simlist$lastnumofmachines = 2
222   simlist$lasttimeup = 0
223   simlist$lasttimedown = 0
224   #Enter main loop. event types and all logic happens via mr1creact function
225   mainloop(simlist, timelim) #note that the last even with time over the time li
226
227   print("total time:")
228   print (simlist$totaltime)
229   print("pi for each state: ")
230   print(simlist$results/simlist$totaltime)
231   return (simlist$results/simlist$totaltime)
232 }
233
234 mr1creact <- function(evnt, simlist) {
235   curtime <- evnt[1]
236   etype <- evnt[2]
237   machnum <- evnt[3]
238
239
240   if (etype == 1){ #machine breaks
241
242     delta_uptime <- curtime - simlist$lasttimeup
243     if (simlist$lastnumofmachines == 2){
244       simlist$lastnumofmachines = 1
245       simlist$results[3] <- simlist$results[3] + delta_uptime
246       simlist$totaltime <- simlist$totaltime + delta_uptime
247     } else { #1 machine was up
248       simlist$lastnumofmachines = 0
```

```
249       simlist$results[2] <- simlist$results[2] + delta_uptime
250       simlist$totaltime <- simlist$totaltime + delta_uptime
251     }
252
253     if (simlist$mr1c[1]==1){ #remairman is offsite
254       ttr <- rexp(1, simlist$lambda_r) #time till repair end
255       waittime <- simlist$c
256
257     } else{ #repairer is on site
258       ttr <- rexp(1, simlist$lambda_r) #time till repair end
259       waittime <- simlist$mr1c[2]
260     }
261
262     simlist$mr1c <- c(2,ttr + waittime) #new waittime till mr1c will be free
263     schedevnt(curtime + ttr + waittime, 2, simlist, c(machnum))
264
265     simlist$lasttimedown <- curtime
266
267 #     print ("--------current time----------")
268 #     print (curtime)
269 #     print ("---machine broke---")
270 #     print (machnum)
271 #     print ("time till fix: ")
272 #     print (ttr + waittime)
273 #     print ("stats - time in each state {0,1,2}")
274 #     print (simlist$results)
275
276   } else { #a machine is repaired
277
278     delta_downtime <- curtime - simlist$lasttimedown
279
280     if (simlist$lastnumofmachines == 0){ #[the downtime of this machine is how
281       simlist$lastnumofmachines = 1
282       simlist$results[1] <- simlist$results[1] + delta_downtime
283       simlist$totaltime <- simlist$totaltime + delta_downtime
284     } else { #1 machine was up [the downtime of this machine is how long we had
285       simlist$lastnumofmachines = 2
286       simlist$results[2] <- simlist$results[2] + delta_downtime
287       simlist$totaltime <- simlist$totaltime + delta_downtime
288     }
289
```

```
290      if (curtime >= simlist$mr1c[2]){ #mr1c is free to go, the other machine is
291        simlist$mr1c <- c(1,0)
292      } else{
293        #nothing changes for the mr1c if he isn't free to go
294      }
295      ttf <- rexp(1, simlist$lambda_u) #time till next failure
296      schedevnt(simlist$currtime + ttf, 1, simlist, c(machnum))
297
298      simlist$lasttimeup <- curtime
299
300 #      print ("------current time--------------")
301 #      print (curtime)
302 #      print ("---machine repaired---")
303 #      print (machnum)
304 #      print ("stats - time in each state {0,1,2}")
305 #      print (simlist$results)
306
307    }
308 }
309
310 ######## end of problem 1c ####
```

Problem 2(b)

```
1
2 # DES.R:  R routines for discrete-event simulation (DES), with an example
3
4 # matrix version; data frame allows character event types, but much too slow
5
6 # all data is stored in an R environment variable that will be referrred
7 # to as simlist below
8
9 # the simlist will consist of the following components:
10 #
11 #       currtime:  current simulated time
12 #       evnts:  the events list, a matrix
13 #       reactevent:  event handler, user-supplied; creates new
14 #                    events upon the occurrence of an old one;
15 #                    e.g. job arrival triggers either start of
16 #                    service for the job or queuing it; call form is
17 #                    reactevent(evnt,simlist)
18 #       dbg:  if TRUE, will print evnts above after each event
```

```
19 #               scheduling action, and enter R browser for single-stepping
20 #               etc.
21
22 # the application code can add further application-specific data to
23 # simlist, e.g. total job queuing time
24
25 # each event will be represented by a matrix row consisting of:
26 #
27 #    occurrence time
28 #    event type (user-defined numeric code)
29 #
30 # and application-specific information, if any
31
32 # library functions (do not alter):
33 #
34 #       newsim:  create a new simlist
35 #       insevnt:  insert a new event into evnts in the simlist
36 #       schedevnt:  schedule a new event (determine its occurrence time
37 #                   and call insevnt())
38 #       getnextevnt:  pulls the earliest event from the event list,
39 #                     process it, and update the current simulated
40 #                     time
41 #       mainloop:  as the name implies
42 #       appendtofcfsqueue:  append job to a FCFS queue
43 #       delfcfsqueue:  delete head of a FCFS queue
44
45 # outline of a typical application:
46
47 #    mysim <- newsim()    create the simlist
48 #    set reactevent in mysim
49 #    set application-specific variables in mysim, if any
50 #    set the first event in mysim$evnts
51 #    mainloop(mysim,mysimtimelim)
52 #    print results
53
54 # create a simlist, which will be the return value, an R environment
55 newsim <- function(dbg=F) {
56    simlist <- new.env()
57    simlist$currtime <- 0.0  # current simulated time
58    simlist$evnts <- NULL  # event list
59    simlist$dbg <- dbg
```

```
60      simlist
61  }
62
63  # insert event evnt into evnts in simlist
64  insevnt <- function(evnt,simlist) {
65      # if the event list is empty, set it to consist of evnt and return
66      if (is.null(simlist$evnts)) {
67          simlist$evnts <- matrix(evnt,nrow=1)
68          return()
69      }
70      # otherwise, find insertion point
71      inspt <- binsearch(simlist$evnts[,1],evnt[1])
72      # now "insert," by reconstructing the matrix; we find what portion of
73      # the current matrix should come before evnt and what portion should
74      # come after it, then string everything together
75      before <- if (inspt == 1) NULL else simlist$evnts[1:(inspt-1),]
76      nr <- nrow(simlist$evnts)
77      after <- if (inspt <= nr) simlist$evnts[inspt:nr,] else NULL
78      simlist$evnts <- rbind(before,evnt,after)
79      rownames(simlist$evnts) <- NULL
80  }
81
82  # schedule new event in evnts in simlist; evnttime is the time at
83  # which the event is to occur; evnttype is the event type; appdata is
84  # a vector of numerical application-specific data
85  schedevnt <- function(evnttime,evnttype,simlist,appdata=NULL) {
86      evnt <- c(evnttime,evnttype,appdata)
87      insevnt(evnt,simlist)
88  }
89
90  # start to process next event (second half done by application
91  # programmer via call to reactevnt() from mainloop())
92  getnextevnt <- function(simlist) {
93      head <- simlist$evnts[1,]
94      # delete head
95      if (nrow(simlist$evnts) == 1) simlist$evnts <- NULL else
96          simlist$evnts <- simlist$evnts[-1,,drop=F]
97      return(head)
98  }
99
100 # main loop of the simulation
```

```
101 mainloop <- function(simlist,simtimelim) {
102    while(simlist$currtime < simtimelim) {
103       head <- getnextevnt(simlist)
104       # update current simulated time
105       simlist$currtime <- head[1]
106       # process this event (programmer-supplied ftn)
107       simlist$reactevent(head,simlist)
108       if (simlist$dbg) {
109          print("event␣occurred:")
110          print(head)
111          print("events␣list␣now")
112          print(simlist$evnts)
113          browser()
114       }
115    }
116 }
117
118 # binary search of insertion point of y in the sorted vector x; returns
119 # the position in x before which y should be inserted, with the value
120 # length(x)+1 if y is larger than x[length(x)]; this could be replaced
121 # by faster C code
122 binsearch <- function(x,y) {
123    n <- length(x)
124    lo <- 1
125    hi <- n
126    while(lo+1 < hi) {
127       mid <- floor((lo+hi)/2)
128       if (y == x[mid]) return(mid)
129       if (y < x[mid]) hi <- mid else lo <- mid
130    }
131    if (y <= x[lo]) return(lo)
132    if (y < x[hi]) return(hi)
133    return(hi+1)
134 }
135
136 # appendtofcfsqueue() and delfcfsqueuehead() below assume the
137 # application code has one or more queues, each queue stored as a
138 # list-of-lists, with each individual list being the information for one
139 # queued job; note that one must initialize the list-of-lists as NULL
140
141 # appends jobtoqueue to the given queue, assumed of the above form;
```

```
142 # the new, longer list is returned
143 appendtofcfsqueue <- function(queue,jobtoqueue) {
144     lng <- length(queue)
145     queue[[lng+1]] <- jobtoqueue
146     queue
147 }
148
149 # deletes head of queue; assumes list-of-lists structure as decribed
150 # above; returns the head and new queue
151 delfcfsqueuehead <- function(queue) {
152     qhead <- queue[[1]]
153     newqueue <- queue[-1]
154     # careful!--an empty list is not NULL
155     if (length(queue) == 1) newqueue <- NULL
156     list(qhead=qhead,newqueue=newqueue)
157 }
158
159
160
161
162 ## Start of our code.
163
164
165 # Nurse Problem
166 # Parameters
167 # n :: limit on active nurses
168 # q :: limit on queue size
169 # p :: timeout time
170 # d :: mean of call duration (exponential)
171 # r :: mean of call arrival (uniform? [0, 2r])
172 # (stated in problem that there is an issue with exponential arrival, but I don
173 # (Is it that it doesn't make sense as a policy if arrival time is exponential?,
174
175
176 cc2b <- function(d, r, n, q, p, timelim, dbg=F) {
177   # Event types:
178   # 1 - call arrived
179   # 2 - call ended
180   # 3 - timeout
181
182   # Set up simulation list, specify event handler.
```

```
183  simlist <- newsim(dbg)
184  simlist$reactevent <- cc2breact
185
186  # Parameters required by factoryreact()
187  simlist$lambda_d = 1/d
188  simlist$r = r #for uniform
189  simlist$lambda_r = 1/r #for exp
190
191  simlist$n <- n # max active nurses
192  simlist$q <- q # max queue size
193  simlist$p <- p
194
195  # initial conditions : one active nurse, no calls in queue
196  simlist$i_n <- 1 # active nurse
197  simlist$i_i <- 1 # idle nurses
198  simlist$i_q <- 0 # queued calls
199
200
201
202
203  # We must generate the first event and handle it.
204  # Since the simulation starts the queue will be empty,
205  # the first event will be a call arriving
206  tta <- runif(1, 0, 2*r)
207
208  # tta <- rexp(1, 1/r) # for exp
209
210  ## Is this necessary? Couldn't you just use the timelim passed to the system?
211  ## Ohhh, right, it's used to tally up the total time each system exists. I can
212
213  simlist$lasttime <- 0.0
214  simlist$activeTime <- 0.0
215  simlist$idleTime <- 0.0
216  schedevnt(tta, 1, simlist)
217
218  # Flag for timeout
219  simlist$reset <- F
220  simlist$nextTimeout <- p
221
222  # The way I'm configuring this, there is also a running timeout event.
223  # This will just continually run, setting the next timeout to the value above.
```

```
224   # so I need to start this event as well.
225
226   if(p > 0){ # only start it if there is a positive timeout value! Otherwise, ig
227         schedevnt(p, 3, simlist)
228   }
229   # Running totals for dropped calls
230   simlist$rej <- 0
231   simlist$tot <- 0
232
233   # Enter main loop (calls factoryreact()).
234   mainloop(simlist, timelim)
235
236   # Report average number of machines running.
237   simlist$time <- simlist$idleTime / simlist$activeTime
238   proRej <- simlist$rej / simlist$tot
239   print("Proportion of calls rejected")
240   print(proRej)
241   print("Proportion of nurse idle time")
242   print(simlist$time)
243 }
244
245 # Our reactevent(). Transition to new state
246 # and generate next event.
247 cc2breact <- function(evnt, simlist) {
248   etype <- evnt[2]
249
250   # Transition state.
251
252   if (etype == 1){ # call arrived
253
254         simlist$reset <- T
255     simlist$nextTimeout <- simlist$currtime + simlist$p
256         simlist$tot <- simlist$tot + 1
257
258     print ("-----Call arrived at time-----")
259     print (evnt[1])
260
261         if(simlist$i_i > 0){ # if idle nurses to take call (queue empty)
262
263               # active time calculations
264
```

```
265             # Essentially, I want to do this calculation whenever the idle c
266             # list changes, for any reason. $lasttime will be set to the las
267
268             delta <- simlist$currtime - simlist$lasttime
269
270             simlist$activeTime <- simlist$activeTime + (simlist$i_n * delta)
271             simlist$idleTime <- simlist$idleTime + (simlist$i_i * delta)
272             simlist$lasttime <- simlist$currtime
273
274             # Now that that's sorted, we can move on
275
276             simlist$i_i <- simlist$i_i - 1
277
278             # new event: call ended
279             tte <- rexp(1, simlist$lambda_d)
280             schedevnt(simlist$currtime + tte, 2, simlist)
281
282     print ("an idle nurse takes the call")
283
284         } else if(simlist$i_q < simlist$q){ # if queue not full, no nurses to
285
286                 simlist$i_q <- simlist$i_q + 1
287
288     print ("no nurse call take the call, queue is not full")
289         } else { # queue full, call dropped, new active nurse
290
291                 simlist$rej <- simlist$rej + 1
292
293                 if(simlist$i_n < simlist$n){ # if nurse limit not reached
294
295                     # active time calculations
296
297                     delta <- simlist$currtime - simlist$lasttime
298
299                     simlist$activeTime <- simlist$activeTime + (simlist$i_n
300                     simlist$idleTime <- simlist$idleTime + (simlist$i_i * d
301                     simlist$lasttime <- simlist$currtime
302
303
304                     # Add one nurse, take call from queue
305                     simlist$i_n = simlist$i_n + 1
```

```
306                         simlist$i_q = simlist$i_q - 1
307
308                         # new event: call ended
309                         tte <- rexp(1, simlist$lambda_d)
310                         schedevnt(simlist$currtime + tte, 2, simlist)
311                 }
312
313       print ("queue is full, grab new nurse, drop top call")
314     }
315
316     print ("Stats so far:")
317     print ("total calls")
318     print (simlist$tot)
319     print ("total rejected calls")
320     print (simlist$rej)
321     print ("active nurses")
322     print (simlist$i_n)
323     print ("idle nurses")
324     print (simlist$i_i)
325     print ("active nurse time")
326     print (simlist$activeTime)
327     print ("idle nurse time")
328     print (simlist$idleTime)
329
330         # new event: call arrival
331         tta <- runif(1, 0, 2*simlist$r)
332         schedevnt(simlist$currtime + tta, 1, simlist)
333
334   } else if (etype == 2) { # call ended
335
336     print ("-----Call ended at time-----")
337     print (evnt[1])
338
339         if(simlist$i_q > 0){ # calls in queue
340
341                 simlist$i_q <- simlist$i_q - 1
342
343                 # new event: call ended
344                 tte <- rexp(1, simlist$lambda_d)
345                 schedevnt(simlist$currtime + tte, 2, simlist)
346
```

```
347       print ("there are calls in queue ")
348
349         } else { # queue empty, new idle nurse
350
351                 # active time calculations
352
353                 delta <- simlist$currtime - simlist$lasttime
354
355                 simlist$activeTime <- simlist$activeTime + (simlist$i_n * delta)
356                 simlist$idleTime <- simlist$idleTime + (simlist$i_i * delta)
357                 simlist$lasttime <- simlist$currtime
358
359                 # new idle nurse
360
361                 simlist$i_i <- simlist$i_i + 1
362
363                 if(simlist$p <= 0 && simlist$i_n > 1){ # if no timeout value and
364                         simlist$i_i <- simlist$i_i - 1
365                         simlist$i_n <- simlist$i_n - 1
366                 }
367
368       print("queue is empty - new idle nurse")
369           }
370
371     print ("Stats so far:")
372     print ("total calls")
373     print (simlist$tot)
374     print ("total rejected calls")
375     print (simlist$rej)
376     print ("active nurses")
377     print (simlist$i_n)
378     print ("idle nurses")
379     print (simlist$i_i)
380     print ("active nurse time")
381     print (simlist$activeTime)
382     print ("idle nurse time")
383     print (simlist$idleTime)
384   } else if (etype == 3) { # timeout
385
386     print ("-----Timeout happened at time-----")
387     print (evnt[1])
```

```
388
389         if(simlist$reset) {
390
391       print ("previously arrived call has reset this arrival")
392                 # timeout has been reset by a call arriving
393                 simlist$reset <- F
394         } else if (simlist$i_n == 1) {
395                 # only one nurse left
396                 # Reset timeout
397                 simlist$nextTimeout <- simlist$currtime + simlist$p
398
399       print ("only one nurse is left, reset timeout") #probably don't even need
400         } else if (simlist$i_i > 0) { # Some idle nurses ( >1)
401                 # active time calculations
402
403                 delta <- simlist$currtime - simlist$lasttime
404
405                 simlist$activeTime <- simlist$activeTime + (simlist$i_n * delta)
406                 simlist$idleTime <- simlist$idleTime + (simlist$i_i * delta)
407                 simlist$lasttime <- simlist$currtime
408
409                 # remove nurses from active pool
410
411                 simlist$i_n <- simlist$i_n - 1
412                 simlist$i_i <- simlist$i_i - 1
413
414                 # reset timeout
415                 simlist$nextTimeout <- simlist$currtime + simlist$p
416       print ("there are idle nurses, one of them is sent to inactive")
417         } else { # no idle nurses
418                 simlist$nextTimeout <- simlist$currtime + simlist$p
419       print ("no idle nurses - reset timer?") # is this correct logic? what if
420         }
421         # new event : next timeout
422
423         schedevnt(simlist$nextTimeout, 3, simlist)
424
425     print ("Stats so far:")
426     print ("total calls")
427     print (simlist$tot)
428     print ("total rejected calls")
```

```
429     print (simlist$rej)
430     print ("active nurses")
431     print (simlist$i_n)
432     print ("idle nurses")
433     print (simlist$i_i)
434     print ("active nurse time")
435     print (simlist$activeTime)
436     print ("idle nurse time")
437     print (simlist$idleTime)
438
439   }
440
441
442   # Okay, here's where my design is different, with DES.
443   # Yes, I know the memoryless property says you can schedule things
444   # such that there is exactly one event rolling. But honestly,
445   # that just obscures intuition to me. The intuition is that all
446   # of these things run in parallel, so I schedule multiple events
447   # at once. Because of the memoryless property, this is equivalent
448   # to scheduling them sequentially.
449
450 }
```