Introduction to Discrete-Event Simulation in R

Norm Matloff

January 28, 2009

Discrete-event simulation (DES) is widely used in business, industry and government. The term "discrete event" refers to the fact that the state of the system changes only at discrete times, rather than changing continuously. A typical example would involve a queuing system, say people lining up to use an ATM machine. The number of people in the queue increases only when someone arrives, and decreases only when a person finishes an ATM transaction, both of which occur only at discrete times.

It is not assumed here that the reader has prior background in DES. For our purposes here, the main ingredient to understand is the *event list*, which will now be explained.

Central to DES operation is maintenance of the *event list*, a list of scheduled events. Since the earliest event must always be handled next, the event list is usually implemented as some kind of priority queue. The main loop of the simulation repeatedly iterates, in each iteration pulling the earliest event off of the event list, updating the simulated time to reflect the occurrence of that event, and reacting to this event. The latter action will typically result in the creation of new events. R's lack of pointer variables means that we must write code for maintaining the event list in a nontraditional way, but on the other hand it will also lead to some conveniences too.

One of the oldest approaches to write DES code is the *event-oriented paradigm*. Here the code to handle the occurrence of one event sets up another event. In the case of an arrival to a queue, the code may then set up a service event (or, if there are queued jobs, it will add this job to the queue). As an example to guide our thinking, consider the ATM situation, and suppose we store the event list as a simple vector.

At time 0, the queue is empty. The simulation code randomly generates the time of the first arrival, say 2.3. At this point the event list is simply (2.3). This event is pulled off the list, simulated time is updated to 2.3, and we react to the arrival event as follows: The queue for the ATM is empty, so we start the service, by randomly generating the service time; say it is 1.2 time units. Then the completion of service will occur at simulated time 2.3+1.2 = 3.5, so we add this event to the event list, which will now consist of (3.5). We will also generate the time to the next arrival, say 0.6, which means the arrival will occur at time 2.9. Now the event list consists of (2.9,3.5).

As will be detailed below, our example code here is hardly optimal, and the reader is invited to improve it. It does, however, serve to illustrate a number of R issues. The code consists of some generally-applicable library functions, such as **schedevnt()** and **mainloop()**, and a sample application of those library functions. The latter simulates an M/M/1 queue, i.e. a single-server queue in which both interarrival time and service time are exponentially distributed.

```
1  # DES.r:  R routines for discrete-event simulation (DES), with an example
2
3  # each event will be represented by a vector; the first component will
4  # be the time the event is to occur; the second component will be the
```

```
# numerical code for the programmer-defined event type; the programmer
# may add application-specific components

# a list named "sim" holds the events list and other information; for
# convenience, sim has been stored as a global variable; some functions
# have side effects

# create "sim"
newsim <- function(numfields) {
   sim <<- list()
   sim$currtime <<- 0.0  # current simulated time
   sim$evnts <<- NULL  # event list
}

# insert event evnt into event list
insevnt <- function(evnt) {
   if (is.null(sim$evnts)) {
      sim$evnts <<- matrix(evnt,nrow=1)
      return()
   }
   # find insertion point
   inspt <- binsearch(sim$evnts[,1],evnt[1])
   # now "insert"
   if (inspt > 1) e <- rbind(sim$evnts[1:(inspt-1),],evnt)
   nrse <- nrow(sim$evnts)
   if (inspt <= nrse)
      e <- rbind(evnt, sim$evnts[inspt:nrse,])
   sim$evnts <<- e
}

# schedule new event; evnttime is the time at which the event is to
# occur; evnttype is the event type; and appfields are the values of the
# programmer-defined fields, if any
schedevnt <- function(evnttime,evnttype,appfields=NULL) {
   evnt <- c(evnttime,evnttype,appfields)
   insevnt(evnt)
}

# start to process next event (second half done by application
# programmer via call to reactevnt())
getnextevnt <- function() {
   head <- sim$evnts[1,]
   # delete head
   if (nrow(sim$evnts) == 1) sim$evnts <<- NULL else
      sim$evnts <<- sim$evnts[-1,,drop=F]
   return(head)
}

# main loop of the simulation
mainloop <- function(maxsimtime) {
   while(sim$currtime < maxsimtime) {
      head <- getnextevnt()
      sim$currtime <<- head[1]  # update current simulated time
      reactevnt(head)  # process this event (programmer-supplied ftn)
   }
}

# binary search of insertion point of y in the sorted vector x; returns
# the position in x before which y should be inserted, with the value
# length(x)+1 if y is larger than x[length(x)]
binsearch <- function(x,y) {
   n <- length(x)
   lo <- 1
   hi <- n
   while(lo+1 < hi) {
      mid <- floor((lo+hi)/2)
      if (y == x[mid]) return(mid)
      if (y < x[mid]) hi <- mid else lo <- mid
```

```
    }
    if (y <= x[lo]) return(lo)
    if (y < x[hi]) return(hi)
    return(hi+1)
}

# application:  M/M/1 queue, arrival rate 0.5, service rate 1.0

# globals
# rates
arvrate <- 0.5
srvrate <- 1.0
# event types
arvtype <- 1
srvdonetype <- 2
# initialize server queue
srvq <- NULL  # will just consist of arrival times of queued jobs
# statistics
njobsdone <- NULL  # jobs done so far
totwait <- NULL  # total wait time so far

# event processing function requried by general DES code above
reactevnt <- function(head) {
   if (head[2] == arvtype) {  # arrival
      # if server free, start service, else add to queue
      if (length(srvq) == 0) {
         srvq <<- head[3]
         srvdonetime <- sim$currtime + rexp(1,srvrate)
         schedevnt(srvdonetime,srvdonetype,head[3])
      } else srvq <<- c(srvq,head[3])
      # generate next arrival
      arvtime <- sim$currtime + rexp(1,arvrate)
      schedevnt(arvtime,arvtype,arvtime)
   } else {  # service done
      # process job that just finished
      # do accounting
      njobsdone <<- njobsdone + 1
      totwait <<- totwait + sim$currtime - head[3]
      # remove from queue
      srvq <<- srvq[-1]
      # more still in the queue?
      if (length(srvq) > 0) {
         # schedule new service
         srvdonetime <- sim$currtime + rexp(1,srvrate)
         schedevnt(srvdonetime,srvdonetype,srvq[1])
      }
   }
}

mm1sim <- function() {
   srvq <<- vector(length=0)
   njobsdone <<- 0
   totwait <<- 0.0
   # create simulation, 1 extra field (arrival time)
   newsim(1)
   # get things going, with first arrival event
   arvtime <- rexp(1,rate=arvrate)
   schedevnt(arvtime,arvtype,arvtime)
   mainloop(100000.0)
   return(totwait/njobsdone)
}
```

The simulation state, consisting of the current simulated time and the event list, have been placed in an R list, **sim**. This was done out of a desire to encapsulate the information, which in R typically means using a list.

This list **sim** has been made a global variable, for convenience and clarity. This has led to the use of R's superassignment operator $<<-$, with associated side effects. For instance in **mainloop()**, the line

```
sim$currtime <<- head[1]  # update current simulated time
```

changes a global directly, while **sim$evnts** is changed indirectly via the call

```
head <- getnextevnt()
```

If one has objections to use of globals, this could be changed, though without pointers it could be done only in a limited manner.

As noted, a key issue in writing a DES library is the event list. It has been implemented here as a matrix, **sim$evnts**. Each row of the matrix corresponds to one scheduled event, with information on the event time, the event type (say arrival or service completion) and any application-specific data the programmer wishes to add. The rows of the matrix are in ascending order of event time, which is contained in the first column.

The main potential advantage of using a matrix as our structure here is that it enables us to maintain the event list in ascending order by time via a binary search operation by event time in that first column. This is done in the line

```
inspt <- binsearch(sim$evnts[,1],evnt[1])
```

in **insevnt()**. Here we wish to insert a newly-created event into the event list, and the fact that we are working with a vector enables the use of a fast binary search.

However, looks are somewhat deceiving here. Though for an event set of size n, the search will be of time order O(log n), we still need O(n) to reassign the matrix, in the code

```
if (inspt > 1) e <- rbind(sim$evnts[1:(inspt-1),],evnt)
nrse <- nrow(sim$evnts)
if (inspt <= nrse)
   e <- rbind(evnt, sim$evnts[inspt:nrse,])
sim$evnts <<- e
```

Again, this exemplifies the effects of lack of pointers. Here is a situation in which it may be useful to write some code in C/C++ and then interface to R.

This code above is a good example of the use of **rbind()**. We use the function to build up the new version of **sim$evnts** with our new event inserted, row by row. Recall that in this matrix, earlier events are stored in rows above later events. We first use **rbind()** to put together our new event with the existing events that are earlier than it, if any:

```
if (inspt > 1) e <- rbind(sim$evnts[1:(inspt-1),],evnt)
```

Then, unless our new event is later than all the existing ones, we tack on the events that are later than it:

```
nrse <- nrow(sim$evnts)
if (inspt <= nrse)
   e <- rbind(evnt, sim$evnts[inspt:nrse,])
```

There are a couple of items worth mentioning in the line

```
sim$evnts <<- sim$evnts[-1,,drop=F]
```

First, note that the negative-subscript feature of vector operations, in which the indices indicate which elements to skip, applies to matrices too. Here we are in essence deleting the first row of **sim$evnts**.

Second, here is an example of the need for **drop**. If there are just two events, the deletion will leave us with only one. Without **drop**, the assignment would then change **sim$evnts** from a matrix to a vector, causing problems in subsequent code that assumes it is a matrix.

The DES library code we've written above requires that the user provide a function **reactevnt()** that takes the proper actions for each event. In our M/M/1 queue example here, we've defined two types of events—arrival and service completion. Our function **reactevnt()** must then supply code to execute for each of these two events. As mentioned earlier, for an arrival event, we must add the new job to the queue, and if the server is idle, schedule a service event for this job. If a service completion event occurs, our code updates the statistics and then checks the queue; if there are still jobs there, the first has a service completion event scheduled for it.

In this example, there is just one piece of application-specific data that we add to events, which is each job's arrival time. This is needed in order to calculate total wait time.