Answers to all questions should go in a copy of the answer sheet posted on Lyceum.

Begin this lab by writing two different arrival process functions and a single service process function, each using the R-default exponential generator `rexp`:

```
library(simEd)   # load the simEd library
myArr1 <- function() { rexp(1, rate = 1)      }
myArr2 <- function() { rexp(1, rate = 11/10)  }
mySvc1 <- function() { rexp(1, rate = 10/9)   }
```

Now run `ssq()` twice (using your, and my, favorite seed of 8675309), passing the first interarrival function for the first run and the second interarrival function for the second run, while passing the same service process function for each and saving the stats into different vectors:

```
output1 <- ssq(maxArrivals = 20, seed = 8675309,
   interarrivalFcn = myArr1,
   serviceFcn = mySvc1, saveAllStats = TRUE, showOutput = FALSE)

output2 <- ssq(maxArrivals = 20, seed = 8675309,
   interarrivalFcn = myArr2,
   serviceFcn = mySvc1, saveAllStats = TRUE, showOutput = FALSE)
```

Now print the interarrival times and service times from `output1`, and then the interarrival and service times from `output 2`.

You should notice that, using seed 8675309, interarrival times do not match across runs even from the start — this is expected because you are using different rates in `myArr1` vs `myArr2`. More importantly, you should notice that between the first and second runs of `ssq`, the service times match for a while, and then diverge.

Question 1: What are the first ten service times from the first run?

Question 2: What are the first ten service times from the second run?

Question 3: For what customer, and what are the different values, where the first service time diverges between the two runs?

So why is there divergence in the service times between the two runs? They are drawn from the same distribution, after all. To investigate, let's now plot for each run the number in the node across time (the "skyline plot"), encompassing the arrival and completion of the first four jobs (some additional jobs may be included in one of the plots).

```
par(mfrow = c(2,1))  # display two rows, one column

indices <- seq_along(output1$numInSystemT[output1$numInSystemT <= 5])
plot(output1$numInSystemT[indices], output1$numInSystemN[indices],
     type = "s", xlim = c(0,5), bty = "n", las = 1)

indices <- seq_along(output2$numInSystemT[output2$numInSystemT <= 5])
plot(output2$numInSystemT[indices], output2$numInSystemN[indices],
     type = "s", xlim = c(0,5), bty = "n", las = 1)
```

As mentioned in class, the function `ssq` does not use the process-oriented world view you used to construct the table in Lecture 1. For that table, whenever customer i arrives, you immediately "generate" a correspond-

ing service time so that you can determine when that ith *process* (customer) will finish. However, `ssq` uses a next-event world view (we will talk much more about this later). With a next-event world view approach, the service time for the ith customer is not generated until $(i-1)$st customer completes service. The key points:

- In a process-oriented approach, times are always generated in the order of: interarrival, service, interarrival, service, interarrival, service, ... (ad infinitum);
- In a next-event approach, the order in which times are generated is dependent on the timing of arrivals and completions-of-service, so that you may have several interarrivals generated in sequence before another service is generated.

By visually inspecting your skyline plots of the number in the system from each of the two runs:

Question 4: In the first run, how many arrivals occur *after* the fourth arrival, but before that fourth customer completes service?

Question 5: In the second run, how many arrivals occur *after* the fourth arrival, but before that fourth customer completes service?

In both runs, this means several more interarrivals are generated *before* the service time for the fifth customer is generated (at the time the fourth customer completes service). In short, the lock-step interarrival-then-service time generation present in the process-oriented approach no longer exists, so using this approach we cannot guarantee that *exactly the same service process* will be seen.

You can determine the actual completion of the fourth customer using the sum of arrival times (the cumulative sum of interarrival times) and the sojourn times:

```
arrivalTimes <- cumsum(output1$interarrivalTimes)
completionTimes <- arrivalTimes + output1$sojournTimes
```

Do this for both runs, and add a corresponding dashed vertical line to each skyline plot showing where the completion time of the fourth customer occurs.

However, we *want* the two different simulations to see exactly the same service process. In this way, we would only be changing one thing at a time from run-to-run — the interarrival times would change, but the service times would be exactly the same for all customers.

We can achieve this goal by using *streams* of random numbers with our random number generator. As we discussed, a pseudorandom number generator creates a virtual circle of random numbers, and the particular sequence of numbers generated is entirely dependent on the choice of initial seed.

We can use this idea to our advantage by appropriately "jumping" back and forth between/among different sections of the circle for our simulation's different stochastic components (e.g., interarrivals vs. service times). However, the jumps should be done carefully so as to avoid overlap between/among sections.

The v* (e.g., `vexp`, `vgamma`, etc.) versions of variate generators available in `simEd` provide the availability of streams, whereas the r* (e.g., `rexp`, `rgamma`, etc.) versions of R-default variate generators do not. Try the following exercise to use `rexp` to generate interarrival and service times:

```
set.seed(8675309)
rexp(1, rate = 1)
rexp(1, rate = 1)
rexp(1, rate = 1)
rexp(1, rate = 10/9)
rexp(1, rate = 10/9)
rexp(1, rate = 10/9)
```

The first three numbers (rate = 1) are interarrivals, the second three (rate = 10/9) are service times.

Question 6: What are the arrival times (rate = 1) generated using `rexp` (no streams)?

Question 7: What are the service times (rate = 10/9) generated using `rexp` (no streams)?

Now interleave the interarrival and service time generations:

```
set.seed(8675309)
rexp(1, rate = 1)
rexp(1, rate = 10/9)
rexp(1, rate = 1)
rexp(1, rate = 10/9)
rexp(1, rate = 1)
rexp(1, rate = 10/9)
```

Question 8: Now what are the arrival times (rate = 1) generated using `rexp` (no streams)?

Question 9: Now what are the service times (rate = 10/9) generated using `rexp` (no streams)?

Now repeat the exercise using `vexp` rather than `rexp`, and use stream 1 for interarrivals and stream 2 for service times:

```
set.seed(8675309)
vexp(1, rate = 1, stream = 1)
vexp(1, rate = 1, stream = 1)
vexp(1, rate = 1, stream = 1)
vexp(1, rate = 10/9, stream = 2)
vexp(1, rate = 10/9, stream = 2)
vexp(1, rate = 10/9, stream = 2)
```

The first three numbers (rate = 1) are interarrivals, the second three (rate = 10/9) are service times.

Question 10: What are the arrival times (rate = 1) generated using `vexp` (with streams)?

Question 11: What are the service times (rate = 10/9) generated using `vexp` (with streams)?

Now interleave the `vexp` interarrival and service time generations:

```
set.seed(8675309)
vexp(1, rate = 1, stream = 1)
vexp(1, rate = 10/9, stream = 2)
vexp(1, rate = 1, stream = 1)
vexp(1, rate = 10/9, stream = 2)
vexp(1, rate = 1, stream = 1)
vexp(1, rate = 10/9, stream = 2)
```

Question 12: Now what are the arrival times (rate = 1) generated using `vexp` (with streams)?

Question 13: Now what are the service times (rate = 10/9) generated using `vexp` (with streams)?

You should notice that the interarrival and service times generated using streams are exactly the same regardless of the order in which they're generated.

Let's now use the idea of streams to address the issue of `ssq` not seeing the same service times whenever we change the arrival process. Redefine your custom arrival and service process functions using `vexp` and streams as follows (compare to your previous definitions):

```
# with streams
myArr1 <- function() { vexp(1, rate = 1,      stream = 1) }
myArr2 <- function() { vexp(1, rate = 11/10,  stream = 1) }
mySvc1 <- function() { vexp(1, rate = 10/9,   stream = 2) }
```

Now rerun the simulation as before, and compare the results:

```
output1 <- ssq(maxArrivals = 20, seed = 8675309,
   interarrivalFcn = myArr1,
   serviceFcn = mySvc1, saveAllStats = TRUE, showOutput = FALSE)

output2 <- ssq(maxArrivals = 20, seed = 8675309,
   interarrivalFcn = myArr2,
   serviceFcn = mySvc1, saveAllStats = TRUE, showOutput = FALSE)
```

Print the interarrival and service times from each run. What do you notice about the interarrival times? The service times?

Question 14: What is the result of the following:

```
sum(output1$interarrivalTimes - output2$interarrivalTimes)
```

Question 15: What is the result of the following:

```
sum(output1$serviceTimes - output2$serviceTimes)
```

Finally, try running `ssq` again twice, using exactly the same interarrival process for both runs but different service processes for each run. You should be able to print the same interarrival times vectors, but different service times vectors.

We will use random number streams much more throughout the semester, as their use facilitates an important technique known as *variance reduction*.

Submitting: Upload your work (R source, the skyline figures saved as a PNG, a PDF of your answer sheet) to Lyceum.