Project 3: Queue Up!

Max Barshay

due by 4pm Friday, June 5

Queueing theory is a branch of probability that often uses Poisson and Markov ideas. You're going to investigate a *single-server queue*, followed by a *multi-server queue*.

Part 1: An M/M/1 queue.

Customers arrive at a drive-thru according to a Poisson process with rate λ_A per minute (A for arrival). The amount of time it takes for the employees to serve a customer follows an Exponential distribution with mean μ_S (S for service), also in minutes.

The tricky part about modeling a queue is that sometimes customers are served immediately, and sometimes the line gets backed up, even in the homogeneous case (which we're assuming throughout this project).

1. Write a function that simulates a single iteration of this queue.

For the nth drive-thru customer, there are several variables:

- A_n = arrival time
- S_n = service time (that's the Exponential variable)
- B_n = time employees begin taking their order
- D_n = departure time

The variable S_n is a duration of time, while all the others are "absolute" time on the clock.

What makes the queue tricky is relating B_n to the other variables — maybe the nth customer gets served immediately, or maybe they have to wait until the previous customer (and everyone before them) has departed.

The inputs to your function should be λ_A (arrival rate), μ_S (mean service time), and t_{max} (i.e., we're studying the queue from $t = 0$ to $t = t_{max}$).

The output of your function should be a data frame. Each row should correspond to one customer. The data frame should have four columns: the simulated values of A_n, S_n, B_n, D_n.

```
MM1 <- function(lambdaA, muS, tmax){
  An <- numeric()
  Sn <- numeric()
  Bn <- numeric()
  Dn <- numeric()
  X.max <- rpois(1, lambda = lambdaA*tmax)
  An <- sort(runif(X.max,0,tmax))
  num.arrivals <- length(An)
  Sn <- rexp(num.arrivals, rate = 1/muS)
```

```
  Dn[1] <- An[1] + Sn[1]
  Bn[1] <- An[1]
  for (i in 1:(num.arrivals - 1)){
    if (Dn[i] > An[i+1]){
      Bn[i+1] <- Dn[i]
    }
    else{
      Bn[i+1] <- An[i+1]
    }
    Dn[i+1] <- Bn[i+1] + Sn[i+1]
  }
  df <- data.frame(round(An, 3), round(Sn, 3), round(Bn, 3), round(Dn, 3))
  colnames(df) <- c("An", "Sn", "Bn", "Dn")
  return (df)
}
```

2. Run your code once with $\lambda_A = 0.25$ customers per minute and $\mu_S = 3$ minutes over $t_{max} = 60$ minutes. Print out the resulting data frame, preferably rounding the values in the table so it doesn't look too messy.

```
twoframe <- MM1(lambdaA = .25, muS = 3, tmax = 60)
twoframe
```

```
##       An    Sn     Bn     Dn
## 1 10.741 6.882 10.741 17.623
## 2 32.880 5.311 32.880 38.191
## 3 36.192 2.957 38.191 41.148
## 4 54.804 1.087 54.804 55.891
## 5 59.600 2.158 59.600 61.758
```

3. Now write a second function to iterate through your original function many times. The inputs should be the same as the first function (to be passed through) plus N = number of iterations. *You must determine* what the outputs should be, based on the next three questions.

```
big_MM1 <- function(N, lambdaA, muS, tmax){
  many.num.arrivals <- numeric()
  many.not.served.imm <- numeric()
  many.time.in.drivethru <- numeric()
  for (i in 1:N){
    df <- MM1(lambdaA, muS, tmax)
    num.arrivals <- nrow(df)
    many.num.arrivals <- c(many.num.arrivals, num.arrivals)
    not.served.imm <- 0
    time.in.drivethru <- 0
    for (j in 1:num.arrivals){
      if (df[j, 1] != df[j, 3]){ # 1 corresponds to An, 3 corresponds to Bn
        not.served.imm <- not.served.imm + 1
      }
      time.in.drivethru <- time.in.drivethru + (df[j, 4] - df[j, 1]) #4 corresponds to Dn, 1 correspond
    }
```

```
    many.not.served.imm <- c(many.not.served.imm, not.served.imm / num.arrivals)
    many.time.in.drivethru <- c(many.time.in.drivethru, time.in.drivethru / num.arrivals)
  }
  return (data.frame(many.num.arrivals, many.not.served.imm, many.time.in.drivethru))
}
```

Use your functions to answer the next set of questions. Use many iterations (at least 1000, preferably 10,000).

4. Focus on a one-hour interval. Calculate a 95% confidence interval for the true mean number of customers they serve during this hour. Does your interval include the correct answer?

```
num_arr <- big_MM1(N = 3000, lambdaA = .25, muS = 3, tmax = 60)[,1]
```

```
t.test(num_arr)$conf.int[1:2]
```

```
## [1] 14.87486 15.15581
```

I am 95% confident that the true mean of number of customers they serve during this hour is between 14.8748556 and 15.1558111.

The correct answer here is (using properties of poisson processes):

$$t_{max} \cdot \lambda_A = 60 \cdot .25 = 15$$

My confidence interval usually contains 15.

5. Estimate the proportion of customers who are *not* served immediately upon arrival.

```
not.served.imm.25 <- big_MM1(N = 3000, lambdaA = .25, muS = 3, tmax = 60)[,2]
```

```
mean(not.served.imm.25)
```

```
## [1] 0.6063971
```

I estimate that 0.6063971 of customers are not served immediately upon arrival on average.

6. Estimate the average length of time a customer spends in the drive-thru line, including (possibly) waiting and then being served.

```
time.in.drivethru.25 <- big_MM1(N = 3000, lambdaA = .25, muS = 3, tmax = 60)[,3]
```

```
mean(time.in.drivethru.25)
```

```
## [1] 7.263975
```

I estimate that a customer spends 7.2639748 minutes in the drive-thru on average.

7. Let's speed up the arrival process to $\lambda_A = 0.4$. Re-run your second function (which, of course, calls your first one). Estimate the proportion of customers who are *not* served immediately upon arrival.

```
not.served.imm.4 <- big_MM1(N = 3000, lambdaA = .4, muS = 3, tmax = 60)[,2]
```

```
mean(not.served.imm.4)
```

```
## [1] 0.8290985
```

With a sped up arrival process, I now estimate that 0.8290985 of customers are not served immediately upon arrival.

8. Using the same output as in the previous question, estimate the average length of time a customer spends in the drive-thru line, including (possibly) waiting and then being served.

```
time.in.drivethru.4 <- big_MM1(N = 3000, lambdaA = .4, muS = 3, tmax = 60)[,3]
```

```
mean(time.in.drivethru.4)
```

```
## [1] 14.65471
```

With a sped up arrival process, I now estimate that a customer will spend 14.6547062 minutes in the drive-thru on average.

9. Compare the answers from the $\lambda_A = 0.25$ case and the $\lambda_A = 0.4$ case. Do the changes make sense? Explain.

Yes, these changes make sense. We would expect that the proportion of customers that are not served immediately, as well as the mean time in the drive through would increase due to the fact that more people are arriving at the drive-thru, but the time it takes to serve each customer is remaining constant. This amounts to a line of customers forming in the drive-thru.

Part 2: An M/M/k queue.

Now imagine an ice cream parlor, with k servers. (Some day, we'll be allowed to visit ice cream parlors again.) Customers arrive at some rate λ_A per minute, and service times are Exponential with mean μ_S. Customers wait in line and then go to the first available server (you know how lines work!).

10. Write a function that simulates a single iteration of this queue.

The inputs should be the same as in Question 1, along with k. The outputs should be the same as they were in Question 1.

Now things *really* get tricky! You'll have to generate some quantities iteratively again. But also, you need to keep track of when each server gets done with their current customer.

There are multiple ways to attack this. One is, track the departure times *from each of the k servers.* B_n is then a function of the customer's arrival time and the *minimum* of the k server departure times. You also need to keep track of which server (1, 2, ... , k) that customer goes to, so that you model the servers properly.

```r
MMK <- function(k, lambdaA, muS, tmax){
  An <- numeric()
  Sn <- numeric()
  Bn <- numeric()
  Dn <- numeric()
  served.by <- numeric()
  X.max <- rpois(1, lambda = lambdaA*tmax)
  An <- sort(runif(X.max,0,tmax))
  num.arrivals <- length(An)
  Sn <- rexp(num.arrivals, rate = 1/muS)
  serve_complete <- rep(0, k)
  for (i in 1:length(An)){
    server <- which.min(serve_complete)
    served.by[i] <- server
    if (serve_complete[server] < An[i]){
      Bn[i] <- An[i]
    }
    else{
      Bn[i] <- serve_complete[server]
    }
    Dn[i] <- Bn[i] + Sn[i]
    serve_complete[server] <- Dn[i]
  }
  df <- data.frame(round(An, 3), round(Sn, 3), round(Bn, 3), round(Dn, 3))
  colnames(df) <- c("An", "Sn", "Bn", "Dn")
  return (df)
}
```

11. Run your code once with $k = 3$ servers, $\lambda_A = 0.4$ customers per minute, and $\mu_S = 5.5$ minutes, over $t_{max} = 120$ minutes. Print out the resulting data frame, preferably rounding the values in the table so it doesn't look too messy.

```r
res <- MMK(k = 3, lambdaA = .4, muS = 5.5, tmax = 120)
res
```

```
##        An     Sn     Bn     Dn
## 1  11.724  7.409 11.724 19.133
## 2  11.970  7.238 11.970 19.208
## 3  12.104  0.405 12.104 12.509
## 4  17.909  2.952 17.909 20.861
## 5  17.934  3.064 19.133 22.196
## 6  22.412 13.546 22.412 35.958
## 7  23.328  7.153 23.328 30.481
## 8  28.720  6.718 28.720 35.438
## 9  29.025  3.282 30.481 33.763
## 10 34.709  4.315 34.709 39.024
## 11 38.704  1.680 38.704 40.384
## 12 39.028 13.073 39.028 52.101
## 13 39.169  2.684 39.169 41.853
## 14 46.989  1.791 46.989 48.780
## 15 48.341  5.537 48.341 53.878
## 16 50.179  3.341 50.179 53.520
```

```
## 17  51.409  5.399  52.101  57.501
## 18  57.865 14.180  57.865  72.045
## 19  64.332  0.594  64.332  64.925
## 20  65.175 17.439  65.175  82.614
## 21  67.032  3.852  67.032  70.884
## 22  67.533  6.672  70.884  77.556
## 23  70.004  3.457  72.045  75.502
## 24  70.730  1.238  75.502  76.740
## 25  81.119 12.028  81.119  93.147
## 26  82.047  5.208  82.047  87.255
## 27  82.428  3.800  82.614  86.414
## 28  85.573  6.696  86.414  93.110
## 29  94.504  0.763  94.504  95.267
## 30  96.106  1.144  96.106  97.249
## 31  97.312  8.347  97.312 105.659
## 32  98.598  0.434  98.598  99.032
## 33 103.127  0.815 103.127 103.942
## 34 105.207  5.751 105.207 110.958
## 35 107.440  0.731 107.440 108.171
## 36 107.986  2.772 107.986 110.758
## 37 108.808  2.070 108.808 110.877
## 38 115.177  9.720 115.177 124.897
```

12. Now write a second function to iterate through your original function many times. The inputs should be the same as the previous function (to be passed through) plus N = number of iterations. *You must determine* what the outputs should be, based on the next four questions.

```r
big_MMK <- function(N, k, lambdaA, muS, tmax){
  many.num.arrivals <- numeric()
  many.time.at.shop <- numeric()
  many.time.before.served <- numeric()
  many.final.customer <- numeric()
  for (i in 1:N){
    df <- MMK(k, lambdaA, muS, tmax)
    num.arrivals <- nrow(df)
    many.num.arrivals <- c(many.num.arrivals, num.arrivals)
    final.customer.id <- which.max(df[,4])
    many.final.customer <- c(many.final.customer, df[final.customer.id, 4])
    time.at.shop <- 0
    time.before.served <- 0
    for (j in 1:num.arrivals){
      time.at.shop <- time.at.shop + (df[j, 4] - df[j, 1]) # 4 corresponds to Dn, 1 corresponds to An
      time.before.served <- time.before.served + (df[j, 3] - df[j, 1]) # 3 corresponds to Bn, 1 corresp
    }
    many.time.at.shop <- c(many.time.at.shop, time.at.shop / num.arrivals)
    many.time.before.served <- c(many.time.before.served, time.before.served / num.arrivals)
  }
  return (data.frame(many.num.arrivals, many.time.at.shop, many.time.before.served, many.final.customer
}
```

Use your functions to answer the next set of questions. Use many iterations (at least 1000, preferably 10,000).

13. Assume the ice cream parlor is empty at 7pm and closes at 9pm, but they serve anyone who got in the door by 9pm. Calculate a 95% confidence interval for the true mean number of customers they serve. Does that match the correct answer?

```
num.arrivals <- big_MMK(N = 3000, k = 3, lambdaA = .4, muS = 5.5, tmax = 120)[,1]
```

```
num.ttest <- t.test(num.arrivals)
num.ttest$conf.int[1:2]
```

```
## [1] 47.92423 48.40843
```

I am 95% confident that the true mean number of customers they serve during these two hours is between 47.9242317 and 48.408435.

The correct answer here is (using properties of poisson processes):

$$t_{max} \cdot \lambda_S = 120 \cdot .4 = 48$$

My confidence interval usually contains 48.

14. Estimate the average length of time a customer spends in the ice cream shop.

```
mean.time.at.shop <- big_MMK(N = 3000, k = 3, lambdaA = .4, muS = 5.5, tmax = 120)[,2]
mean(mean.time.at.shop)
```

```
## [1] 8.014462
```

15. Estimate the average length of time a customer spends standing in line waiting to be served.

```
mean.time.before.served <- big_MMK(N = 3000, k = 3, lambdaA = .4, muS = 5.5, tmax = 120)[,3]
mean(mean.time.before.served)
```

```
## [1] 2.506465
```

16. On average, at what time do the employees finish serving all the customers? Give a 95% confidence interval for the true expected time that the last customer leaves the ice cream parlor.

```
final.customer.time <- big_MMK(N = 3000, k = 3, lambdaA = .4, muS = 5.5, tmax = 120)[,4]
final.ttest <- t.test(final.customer.time)
final.ttest$conf.int[1:2]
```

```
## [1] 129.7930 130.4481
```

I am 95% confident that true mean time the last customer leaves the parlor is between 129.793008 and 130.4481494.

17. Create a *visualization* for the random process $X(t)$ = number of customers in the ice cream parlor at time t, from $t = 0$ to $t = t_{max}$.

Use the same parameters as above, and just graph a single iteration using the output of your function from Question 10. I'm envisioning something like Figure 7.17 on p. 523 of PAEST, except this $X(t)$ can go up and down.

You'll have to figure out how to deduce the value of $X(t)$ from the information in your data frame. :)

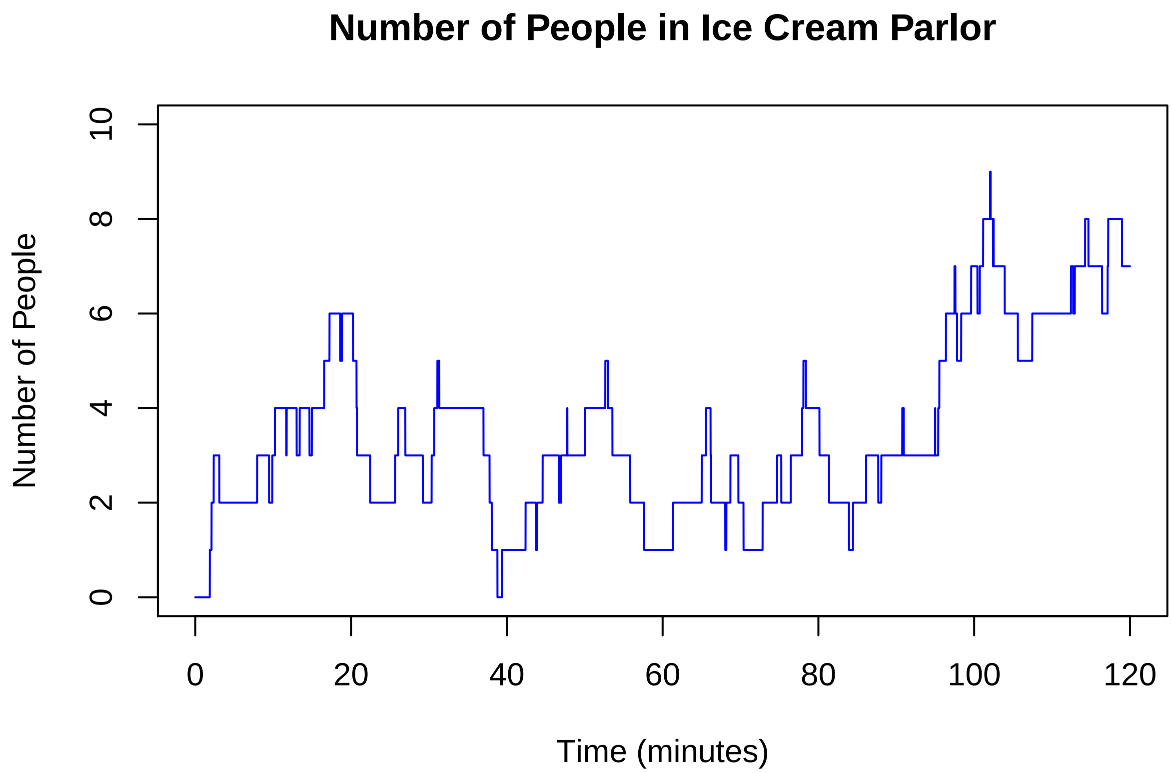
```
X.t.tracker <- function(k, lambdaA, muS, tmax, incr = .01){
  df <- MMK(k, lambdaA, muS, tmax)
  t <- seq(0, tmax, by=incr);
  X.t <- rep(0, length(t))
  for (i in 1:length(t)){
    X.t[i] <- (sum(df[,1] <= t[i]) - sum(df[,4] <= t[i])) # this is the number of people that are in th
  }
  return (list(X.t, t))
}
```

```
X.t <- X.t.tracker(k = 3, lambdaA = .4, muS = 5.5, tmax = 120)
max(X.t[[1]])
```

```
## [1] 9
```

```
simpleplotter <- function(x, y){
  return (plot(x, y,type="l",main="Number of People in Ice Cream Parlor", xlab = "Time (minutes)", ylab=
}
```

```
simpleplotter(X.t[[2]], X.t[[1]])
```



18. At the risk of creating more grading for myself... Use your M/M/k queue simulator to explore at least one other aspect!

This could be the effect of one or more of the parameters, for example, or modeling a new variable. You could change the rules and say anyone not served by 9pm gets kicked out without ice cream. Whatever you think would be interesting. Dazzle me!

I am now going to make it so that anybody who hasn't gotten their ice cream by 9PM gets KICKED OUT (even after having paid). A true tragedy. I am going to simulate the random variable M = # of customers that get kicked out without getting their ice cream.

```
kicked.out.sim <- function(N, k, lambdaA, muS, tmax){
  many.kicked.out <- numeric()
  for (i in 1:N){
    df <- MMK(k, lambdaA, muS, tmax)
    num.arrivals <- nrow(df)
    kicked.out <- 0
    for (j in 1:num.arrivals){
      if (df[j, 4] > tmax){
        kicked.out <- kicked.out + 1
      }
    }
    many.kicked.out <- c(many.kicked.out, kicked.out)
  }
  return (many.kicked.out)
}
```

```r
kicked.out <- kicked.out.sim(3000, k = 3, lambdaA = .4, muS = 5.5, tmax = 120)
```

```r
mean(kicked.out)
```

```
## [1] 3.606667
```

3.6066667 people are tragically unable to get their ice cream on average using these variables.

Let me crank things up a notch and consider a very busy day at the ice cream parlor.

```r
kicked.out.extreme <- kicked.out.sim(3000, k = 1, lambdaA = 10, muS = 10, tmax = 200)
```

```r
mean(kicked.out.extreme)
```

```
## [1] 1979.161
```

1979.1606667 people are unable to get their ice cream on average. In this case the store is likely overrun with only one person working.

Now lets consider an empty store:

```r
kicked.out.empty <- kicked.out.sim(3000, k = 10, lambdaA = 1, muS = 1, tmax = 200)
```

```r
mean(kicked.out.empty)
```

```
## [1] 1.002333
```

In this case only 1.0023333 people are kicked out on average, and this is likely due to the fact that they just arrived late.

Finally, let me two impossible cases just for fun:

```r
kicked.out.ridiculous <- kicked.out.sim(3000, k = 1, lambdaA = 30, muS = 30, tmax = 200)
```

```r
mean(kicked.out.ridiculous)
```

```
## [1] 5993.576
```

Now 5993.5763333 people are kicked out of the parlor.

```r
kicked.out.absurd <- kicked.out.sim(3000, k = 3000, lambdaA = .1, muS = .1, tmax = 200)
```

```r
mean(kicked.out.absurd)
```

```
## [1] 0.007666667
```

Now 0.0076667 people are kicked out of the parlor on average.