ECE 358 S20

M/M/1 and M/M/1/K Queue Simulation

Lab 1

Arjun Bawa, 20711916
Mayuka Bulathsinghala, 20608116
6-1-2020

Contents

Question 1

Our code generated the following experimental results for the mean and variance of 1000 exponential random variables

Mean	Variance
0.0137490037457974	0.000184033045328294

For an exponential random variable, the mean is $1/\lambda$. For $\lambda = 75$, this is $0.01\overline{33}$.

The percentage error between this and the experimental value is $\frac{|0.013749-0.013333|}{0.013333} = 3.12\%$

For an exponential random variable, the variance is $1/\lambda^2$. For $\lambda = 75$, this is $0.000\overline{177}$.

The percentage error between this and the experimental value is $\frac{|0.000184-0.000177|}{0.000177} = 3.95\%$

The percent errors for these two values is small enough that our exponential random variable generator code won't negatively impact the remainder of the experiment.

Question 2

```
from math import log
import csv
import heapq
import random

TRANS_RATE = 1E6 # C := transmission rate of output link in bps, default=1Mbps
SIM_TIME = 1000 # T := simulation time in seconds
AVG_PKT_LEN = 2000 # L := avg len of pkt in bits

# headers meant for csv tables
TITLES = ["Queue_Util", "N_a", "N_d", "N_o", "P_idle", "E[N]"]
TITLES_K = ["Queue_Util", "Buff_size",
      "N_a", "N_d", "N_o", "P_idle", "E[N]", "P_loss"]
```

Figure 1 Defining constants in the code

Our code modularizes most of the sub-functionality of the M/M/1 and M/M/1/K simulators. For a lot of tasks like generating packets, computing statistics and writing to csv files we wrapped the sub-routines into helper functions. This way the logic is easily reusable and configurable using function parameters. It also enabled easy unit testing of the code to make sure each component worked correctly.

Shown in Figure 1 are some constants we defined for the simulations and helper functions to access. Defining these values in once place let us run the large simulations with different parameters easily (such as with longer simulation time). Titles for the csv files were also defined here to simplify writing results to the files.

Packet Generation

expn_random function

```
# Exponential random number generator
def expn_random(rate):
    return (-(1/rate) * log(1 - random.uniform(0, 1)))
```

Figure 2 Helper function that generates an exponential random number from a rate parameter

question1 function

```
# Q1
def question1(f_name='q1.csv', w_type='w'):
    print('-'*10)
    print('Question 1:')
    lam = 75 # lambda
    # list comprehension of 1000 exponential
    #   random vars based on lambda
    randomvars = [expn_random(lam) for i in range(1000)]
    mean = sum(randomvars)/len(randomvars)
    variance = sum( [(r - mean) ** 2 for r in randomvars] ) / len(randomvars)
    print('Mean',mean)
    print('Variance',variance)
    print('-'*10)
    with open(f_name, w_type, newline='') as f:
        w = csv.writer(f, dialect='excel', delimiter=',')
        w.writerow(['Mean','Variance'])
        w.writerow([mean,variance])
```

Figure 3 Function to encapsulate generation of random variables, computing statistics on them and saving the results

gen_ functions

```
# Create arrival events
def gen_arrivals(rate):
    arrival_events = []
    t = expn_random(rate) # time of first pkt arrival
    while t < SIM_TIME:
        arrival_events.append({'time':t,'type':'arrival'})
        t += expn_random(rate)
    return arrival_events
```

Figure 4 Helper function to generate arrival events based on rate of arrival and simulation time. Stores events as list of dictionaries

```
# Create observer events
def gen_observers(arrival_rate):
    observer_events = []
    ''' arrival_rate*(5+random.uniform(0,2)) is used because
    observer events must be at minimum 5 times the rate of
    arrival/departure events. '''
    t = expn_random(arrival_rate*(5+random.uniform(0,2)))
    while t < SIM_TIME:
        observer_events.append({'time':t,'type':'observation'})
        t += expn_random(arrival_rate*(5+random.uniform(0,2)))
    return observer_events
```

Figure 5 Helper function to generate observer events based on rate of arrival and simulation time. Stores events as list of dictionaries

```
# Create departure events
def gen_departures(arrival_events):
    prev_d_time = 0
    departure_events = []
    for ae in arrival_events:
        service_time = gen_service_time()
        d_time = 0
        if (ae['time'] > prev_d_time):
            ''' if the current pkt arrived after the previous pkt
                already left (ie. the queue is empty/idle) '''
            d_time = ae['time']+service_time
        else:
            d_time = prev_d_time + service_time
        departure_events.append({'time':d_time,'type':'departure'})
        prev_d_time = d_time
    return departure_events
```

Figure 6 Function to generate departure events based on arrival events. Works in similar fashion to gen_arrival and gen_observer

```
# Generate service time
def gen_service_time():
    pkt_len = expn_random(1/AVG_PKT_LEN)
    service_time = pkt_len/TRANS_RATE
    return service_time
```

Figure 7 Helper function to generate random service times for packets

```python
# Generate all events for M/M/1 by default, M/M/1/K if K > 0
def gen_events(rate, K=0):
    events = []
    if K == 0:
        # Infinite queue case
        a = gen_arrivals(rate)
        events = sorted(a+gen_departures(a)+gen_observers(rate),
            key=lambda e: e['time'])
    else:
        # Finite queue case
        a = gen_arrivals(rate)
        events = sorted(a+gen_observers(rate), key=lambda e: e['time'])
    return events
```

Figure 8 Function to aggregate the generation of arrival, departure and observer events. Based on whether it's used for the M/M/1 or M/M/1/K simulation, it will either include or exclude the generation of departure events

The set of functions prefixed with "gen" are helper functions meant to streamline generation of events. The separate functions used to generate arrival and observer events are similar in that they both use an arrival rate to sequentially generate respective events at random contiguous times. The gen_observer function scales the rate argument before using it because observation events need to occur at least 5 times as frequently as arrival or departure events for accurate experimental data.

The gen_departure function works a bit differently as it creates departure events based on a list of arrival events passed into the function. This is because each packet has an arrival and departure event, and the departure event for every packet must happen sometime after its arrival depending on the packet's service time. The departure time for each packet in the arrival_events list depends on the departure time of the previous packet, so we keep track of the previous packet's departure time. We can determine the intermediate state of the queue based on this and the current packet's arrival time (i.e. whether the queue is empty upon a new packet's arrival or not). If the queue is not empty, the departure time for a packet is its service time plus the previous packet's departure time. If the queue is empty, then its just the current (arrival) time plus service time. Through this loop we build up a list of valid departure events.

To combine the different event generators, we made the gen_events function. It encapsulates calling each of the specialized event generating functions and aggregates the lists returned by each. Furthermore, the queue buffer size can be passed in as an argument, and the function will return the correct list of events depending on the simulation situation. In the M/M/1 case, the buffer size is infinite so departure events can be calculated before the simulation. This isn't the case in the M/M/1/K simulation, and this function leaves out departure events accordingly. It also sorts the events based on time. We decided to represent events as simple dictionaries in python, consisting of a "time" and "type" field. Dictionaries are not inherently comparable and so aren't sortable. Therefore the sorting of the events list uses a lambda function to specify that the events must be sorted by their numerical "time" field.

M/M/1

simulateMM1 function

```
# Simulate M/M/1
def simulateMM1(q_util):
    pkt_type_count = {
        'arrival':0, # N_a
        'departure':0, # N_d
        'observation':0 # N_o
    }
    idle_count = 0
    current_queue_length = 0
    q_len_observed_over_time = []
    arrival_rate = q_util*TRANS_RATE/AVG_PKT_LEN
    # the 'source' where 'the next packet' is grabbed
    event_list = gen_events(arrival_rate)
    ''' 'q' represents the queue where packets arrive at and depart from.
        We don't need an actual structure for it in this
        case since its size is infinite. '''
    for pkt in event_list:
        # What type of event is it? count it
        pkt_type_count[pkt['type']]+=1
        if pkt['type']=='arrival': current_queue_length+=1
        elif pkt['type']=='departure': current_queue_length-=1
        else:
            # an observer event. observe q_len and save that info
            q_len_observed_over_time.append(current_queue_length)
            # if q empty right now, its idle
            if current_queue_length==0: idle_count+=1
    # P_idle := how often was the queue empty out of the
    #   total times we checked it?
    P_idle = idle_count/pkt_type_count['observation']
    # time average of # of packets in queue; E[N]
    TIME_AVG_PKTS_IN_Q =
        sum(q_len_observed_over_time)/len(q_len_observed_over_time)
    return {TITLES[0]:q_util,
            TITLES[1]:pkt_type_count['arrival'],
            TITLES[2]:pkt_type_count['departure'],
            TITLES[3]:pkt_type_count['observation'],
            TITLES[4]:P_idle,
            TITLES[5]:TIME_AVG_PKTS_IN_Q}
```

Figure 9 The simulation of the M/M/1 queue. Returns the computed statistics from the simulation

This function completely encapsulates the M/M/1 queue simulator using a ρ value passed in as the queue utilization parameter. The pkt_type_count is a simple dictionary that has fields to keep a count of each type of event as they occur in the simulation. The idle_count variable is used to keep a count of how many times we observe the queue to be empty at observer events. The q_len_observed_over_time list keeps a record of the current_queue_length whenever an observer event occurs. The arrival_rate is calculated based on the relation $\lambda = \frac{\rho C}{L}$. Here q_util is the passed-in λ, the TRANS_RATE constant is C and AVG_PKT_LEN is L. An event_list is generated to hold all the events in the simulation and then each event in the list is popped and processed in order.

pkt_type_count[pkt['type']]+=1 increments the correct counter in the dictionary of counters defined above. This works because the "type" field for the generated events is the same string as the names of the fields in pkt_type_count. We're keeping track of the current_queue_length so its incremented on arrival events and decremented on departure events (since arrival means a packet is joining the queue and departure means a packet is leaving the queue). If an event isn't arrival or departure, its an observation event. In this case we take stock of the current_queue_length by appending it to our q_len_observed_over_time list. If the current_queue_length is empty, that means the queue is idle at that moment and we increment the corresponding counter.

P_{idle} is defined as the ratio of times the queue was empty to the total number of times we observed it, in the case of this simulation. So, we computer P_idle according to that ratio. TIME_AVG_PKTS_IN_Q (or $E[N]$) is the average number of packets in the queue during the simulation. We only know the number of packets in the queue every time we observed it, so we sum up all the different lengths of the queue during the simulation and then divide it by how many times we sampled the simulation for its length to compute the average number of packets in the queue. We could have divided by pkt_type_count['observation'] as well, but pkt_type_count['observation'] is equal to len(q_len_observed_over_time) in this case.

We return these results as a dictionary of fields by the names of the TITLES we specified earlier. This makes generating the csv files later much simpler and assures predictable naming conventions of data.

question3 and question4 functions

```python
# Q3
def question3(f_name='q3.csv', w_type='w'):
    print('-'*10)
    print('Question 3:')
    # 0.25 through 0.95
    q_util_list = [i/100 for i in range(25,105,10)]
    # hold results returned from simulation iterations
    results = []
    for i in q_util_list:
        results.append(simulateMM1(i))
        print('~'*10)
        for t in TITLES:
            print(t,results[-1][t])
    print('-'*10)
    with open(f_name, w_type, newline='') as f:
        w = csv.writer(f, dialect='excel', delimiter=',')
        w.writerow(TITLES)
        for r in results:
            w.writerow([r[t] for t in TITLES])
```

Figure 10 Function to encapsulate running the M/M/1 simulator with the range of queue utilization/traffic intensity values

Much like the question1 function, the question3 function above and the question4 function below are meant to encapsulate the experiments from their respective questions in the lab. question3 generates a list of queue utilization values according to the lab manual (using list comprehension for compactness), calls simulateMM1 in a loop with each value in q_util_list and appends the return value (a dictionary) to a results list. The inner loop to print the results essentially iterates through the values in each return result and prints the corresponding label and value pair. Every one of our functions prefixed with "question" functions in a very similar manner.

```python
# Q4
def question4(f_name='q4.csv', w_type='w'):
    print('-'*10)
    print('Question 4:')
    q_util = 1.2
    result = simulateMM1(q_util)
    for t in TITLES:
        print(t,result[t])
    print('-'*10)
    with open(f_name, w_type, newline='') as f:
        w = csv.writer(f, dialect='excel', delimiter=',')
        w.writerow(TITLES)
        w.writerow([result[t] for t in TITLES])
```

Figure 11 Similar to the question3 function, this function runs the M/M/1 simulation with just one queue utilization value, 1.2

Question 3

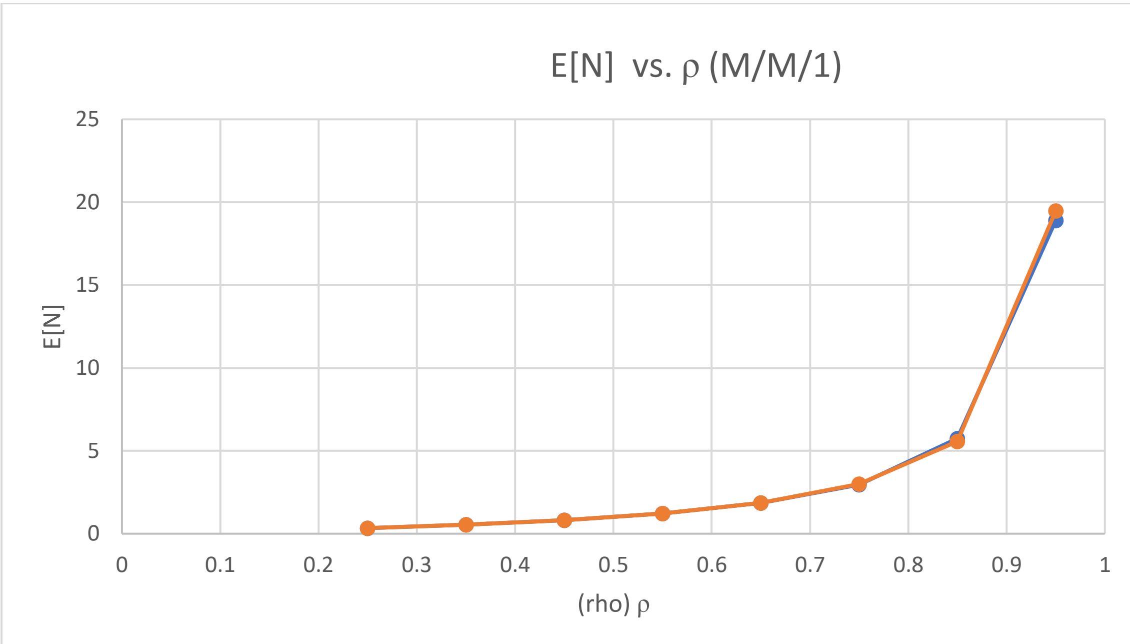

Figure 12 Graph showing the trend of average number of packets in queue (E[N]) with variation in traffic intensity/queue utilization. Shows results from simulation time T = 1000 (blue) and T = 2000 (red)

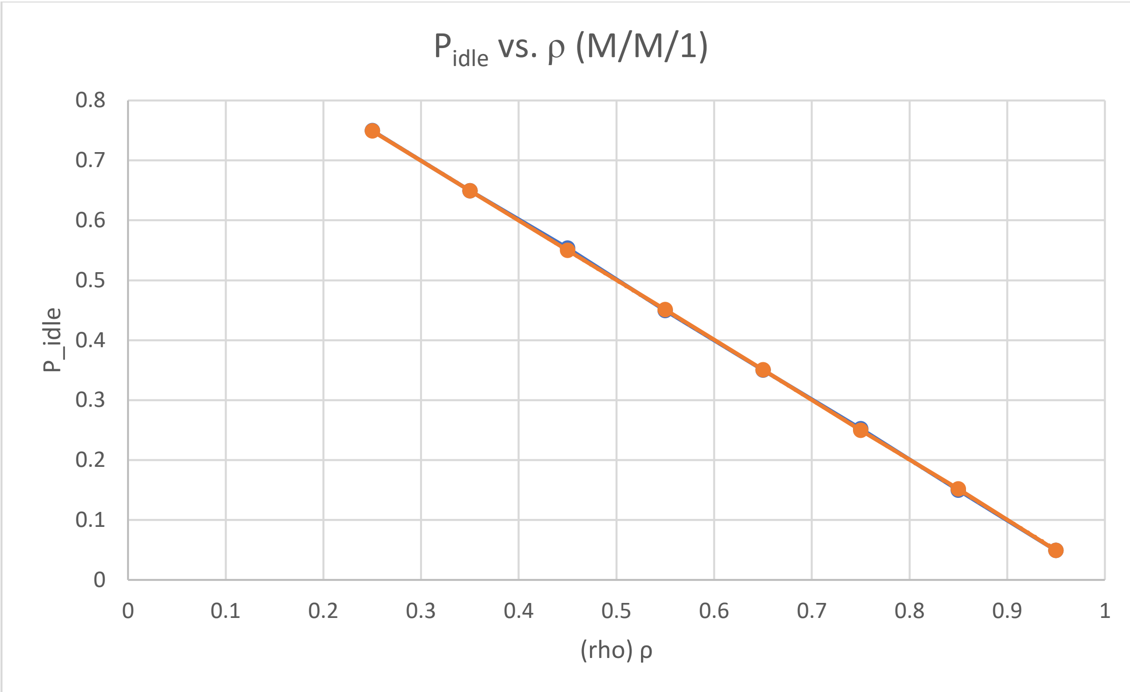

Figure 13 Graph showing the trend of probability of an idle server with variation in traffic intensity/queue utilization. Shows results from simulation time T = 1000 (blue) and T = 2000 (red)

From these graphs we can see that as traffic intensity increases the longer a packet is likely to wait in queue and the less likely it is for the server to be idle.

We obtained these results using the question3 function, which generates list of queue utilization values, calls simulateMM1 in a loop with each value in q_util_list and appends the return value (a dictionary) to a results list. The dictionary stores the E[N] and P_{idle} values.See question 2 for more details. E[N] is the number of packets in the queue and P_{idle} is the percentage of times the server was idle during an observation.

The trendlines for T=1000 and T=2000 are very similar in both graphs. The data from T=1000 and T=2000 is within 5%, thus the data is stable.

Question 4

For ρ=1.2, T=1000, our simulation returns E[N] = 50707 and P_{idle} = 5.61E-07.

For ρ=1.2, T=2000, our simulation returns E[N] = 99368 and P_{idle} = 6.59E-06.

We observe that as the queue_utilization increases, especially over 1, the packets in queue dramatically increase and the time that the server is idle dramatically decreases. The rate at which packets are arriving exceed the rate at which they can be serviced.

Question 5

M/M/1/K

simulateMM1K function

```
# Simulate M/M/1/K
def simulateMM1K(q_util, K):
    pkt_type_count = {
        'arrival':0, # N_a
        'departure':0, # N_d
        'observation':0 # N_o
    }
    idle_count = 0
    q_len_observed_over_time = []
    current_queue_length = 0
    pkts_lost_count = 0
    prev_d_time = 0
    arrival_rate = q_util*TRANS_RATE/AVG_PKT_LEN
    event_list = gen_events(arrival_rate, K)
    # converts events stored as dictionaries to Event
    #   objects for use with heapq
    event_list = [Event(e['time'],e['type']) for e in event_list]
    # an initial heapifying of the event list,
    #   maintaining the heap invariant: event time
    heapq.heapify(event_list)
    while len(event_list) > 0:
        pkt = heapq.heappop(event_list)
        if pkt.type=='arrival':
            serv_time = gen_service_time()
            if current_queue_length < K:
                d_time = 0
                if current_queue_length > 0:
                    d_time = prev_d_time + serv_time
                else:
                    d_time = pkt.time + serv_time
                prev_d_time = d_time
                heapq.heappush(event_list,Event(d_time,'departure'))
                pkt_type_count[pkt.type]+=1
                current_queue_length+=1
            else:
                pkts_lost_count+=1
        elif pkt.type=='departure':
            current_queue_length-=1
            pkt_type_count[pkt.type]+=1
        else:
            pkt_type_count[pkt.type]+=1
```

```
            # an observer event. observe q_len and save that info
            q_len_observed_over_time.append(current_queue_length)
            # if q empty right now, its idle
            if current_queue_length==0:
                idle_count+=1
    P_idle = idle_count/pkt_type_count['observation']
    TIME_AVG_PKTS_IN_Q
      = sum(q_len_observed_over_time)/len(q_len_observed_over_time)
    # P_loss := ratio of packets lost to total packets attempting to arrive
    P_loss = pkts_lost_count/(pkt_type_count['arrival']+pkts_lost_count)
    return {TITLES_K[0]:q_util,
            TITLES_K[1]:K,
            TITLES_K[2]:pkt_type_count['arrival'],
            TITLES_K[3]:pkt_type_count['departure'],
            TITLES_K[4]:pkt_type_count['observation'],
            TITLES_K[5]:P_idle,
            TITLES_K[6]:TIME_AVG_PKTS_IN_Q,
            TITLES_K[7]:P_loss}
```

The simulateMM1K function is very similar to its simulateMM1 counterpart but has some key differences because it simulates a finite queue. It accepts a K parameter as the buffer size for the queue and uses it to call gen_events for the finite queue case. We now keep track of the number of packets lost in pkts_lost_count, and the departure time for a previous packet during the simulation prev_d_time.

This simulation is very computationally intensive in terms of its memory usage and execution time (relative to the simulateMM1 function). Using the sorted python function to re-sort the event_list after each append of a departure event resulted in the simulateMM1K function unable to terminate in a reasonable amount of time. This is because the sorted function has linear time complexity, so that would make the outer loop quadratic time complexity in its worst case. When the number of events is greater than a few million (as is the case in this simulation) each iteration takes on order of a tenth of a second to execute (measured during development with the time module in python). Iterating millions of times with this execution time means the function will terminate in error long before it would've finished the simulation. For this reason, we employed the use of the heapq module in python. It provides efficient methods to maintain priority queue ordering as a heap data structure. In our simulation, the time of events is the heap invariant and adding an event to the queue must maintain the time ordering of the queue's events.

However, heapq requires its elements to be comparable to maintain a priority queue ordering. As we discussed before, dictionaries are inherently incomparable to each other. The solution to this was creating a wrapper Event class (shown below) to represent each event and manually defining the comparison operator for an object of this type. The event_list = [Event(e['time'],e['type']) for e in event_list] converts the dictionary structure of each event in event_list to an Event object. Before we start the simulation loop, we call heapify on this list to ensure correct initial ordering. This one-time call to a function with $log\ n$ time complexity isn't really required (since gen_events returns an ordered list), but it doesn't make any significant difference to the runtime of simulateMM1K.

For the simulation, we loop until the queue is empty. After popping the next event in the queue, we determine the type of event. When the event is of type "departure", we simply decrement the current_queue_length and increment an occurrence of this type of event. If the event is an observation, we follow the same stock-keeping as in the simulateMM1 function.

In the case of an arrival event we have a couple of different scenarios. If the queue is already full (current_queue_length ≥ K) then we drop the arriving packet (and increment the corresponding counter). When the queue still has room, we need to create a corresponding departure event for this arrival event and put it back into the event_list. We do this with the same logic as in the gen_departures function. The key time-saving step here is heappush, as it's an efficient way of putting a new departure event back into the event_list without disrupting the ordering of events.

The new metric being computed in this function is P_loss. It is the ratio of the number of packets lost to the total number of arrival events (regardless of whether they were dropped or not). pkt_type_count['arrival'] only counted the number of packets that *successfully* joined the queue, so we must add it to the number of packets lost to get the total arrival events.

Event class

```python
''' a wrapper class for event time and type.
    Used for M/M/1/K because heapq needs comparable objects.
    The dictionary approach from M/M/1 using a lambda to sort by time
    doesn't work because heapq doesn't have an option to specify
    a comparator key '''
class Event:
    def __init__(self,etime,etype):
        self.time=etime
        self.type=etype
    # the __lt__ (less than) function defines
    #   comparison for this wrapper class
    def __lt__(self, value):
        self.time<value.time
```

Figure 14 Wrapper class for representing an event. Used in the M/M/1/K simulator.

question 6 function

```
# Q6
def question6(f_name='q6.csv', w_type='w'):
    print('-'*10)
    print('Question 6:')
    # 0.5 through 1.5
    q_util_list = [i/100 for i in range(50,160,10)]
    # buffer sizes
    K_list = [10,25,50]
    results=[]
    for q_util in q_util_list:
        for K in K_list:
            results.append(simulateMM1K(q_util,K))
            print('~'*10)
            for t in TITLES_K:
                print(t,results[-1][t])
    print('-'*10)
    with open(f_name, w_type, newline='') as f:
        w = csv.writer(f, dialect='excel', delimiter=',')
        w.writerow(TITLES_K)
        for r in results:
            w.writerow([r[t] for t in TITLES_K])
```

Figure 15 Function to encapsulate running the M/M/1/K simulator with 3 different buffer sizes for each queue utilization/traffic intensity value

Question 6

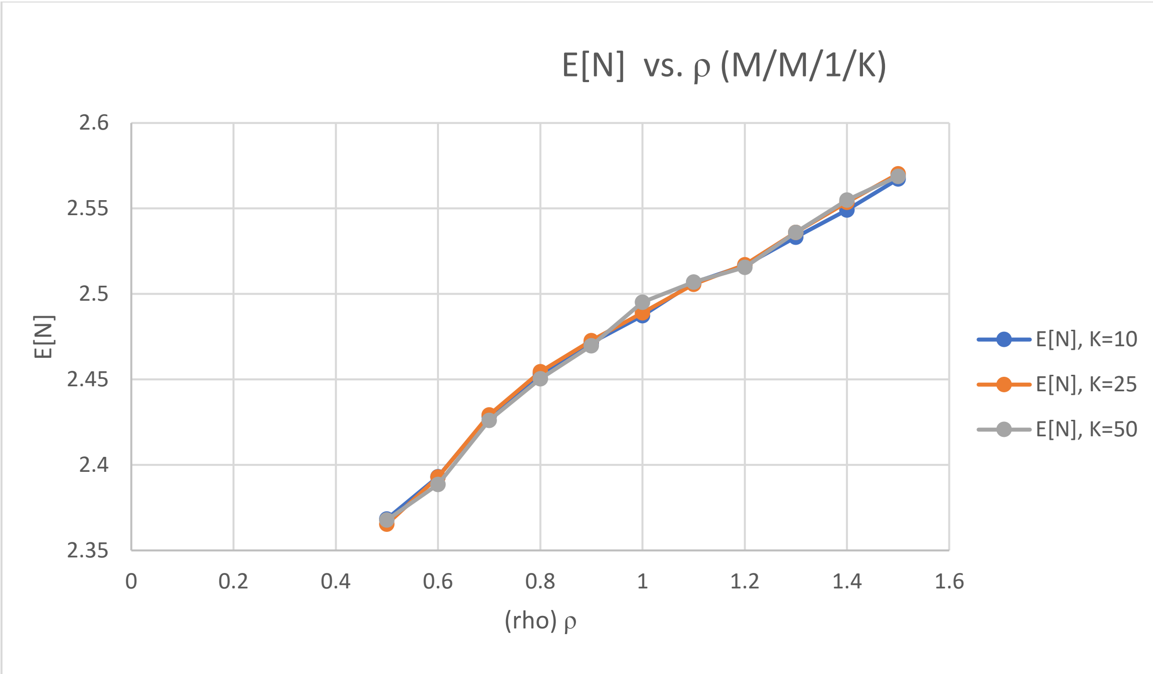

Figure 16 Graph showing the trend of average number of packets in queue (E[N]) with variation in traffic intensity/queue utilization. Shows results with buffer size K=10, 25, 50.

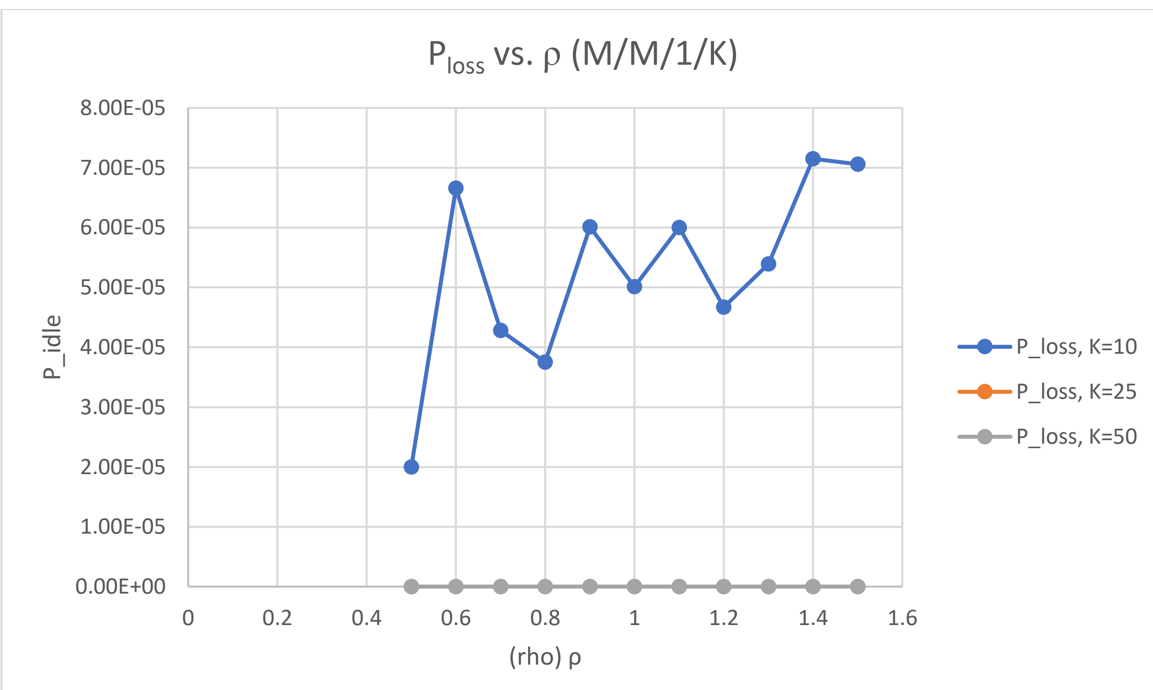

Figure 17 Graph showing the trend of probability of packet loss with variation in traffic intensity/queue utilization. Shows results with buffer size K=10, 25, 50.

P_{loss} was obtained by recording the ratio of packets that arrived when the buffer was full.

Note that for T=1000 and T=2000 the results are stable. As we can see from Figure 17, the number of packets in the queue increase as the traffic intensity increases, as expected. This number is unaffected by the buffer size. We can see, however, that packet loss increases in as the traffic intensity increases, but dramatically decreases as the buffer size decreased. For K=25 and K=50 there was no packet loss, despite higher traffic intensities because the buffer size was able to hold the packets in queue for the time duration.

The q6.csv file includes all the data for T=1000 and T=2000