Table of Contents:

Question 1

-Theory on how you are implementing it:

According to the lab manual, the formula of x = -(1/λ)ln(1-U) can be used to simulate x as the exponential random variable. In our code, we use the random library and random.exponential(1/75) with λ= 75 to generate it. A list is used to record all the generated values for further calculation of the mean and variance.

-Experiment data

mean	variance
0.012938020568775157	0.00019173210734111512
0.013782400193812518	0.00017846530560327206
0.014051095010288499	0.00018519577095978855
0.013191639242620403	0.00016361262279475814
0.012858747653341109	0.0001600107128995934
0.012908278360393355	0.00017949199984793022
0.013471119624210513	0.0001644251149701301
0.013641544775809837	0.00020260038384258363
0.013625975689231056	0.0001809570167673896
0.013285599568457295	0.00017854429929659973
avg	avg
0.013375442068693974	0.00017850353343231605

-The expected value and the variance of an exponential random variable with λ=75 are:

- mean = 1/λ = 1/75 =0.0133
- variance = E[X^2]-(E[X])^2 = 1/λ^2 = 1/75^2=0.00017

- which are consistent with the experimental results

Question 2

-How to implement the queue:

1) In our code, we use variables L, C, rho and lamb to represent the average length of a packet in bits, the transmission rate of the output link in bits per second, and utilization of the queue, respectively. lamb is for the average number of packets arrive, it is generated based on rho*C/L. T is for simulation time, and K indicates the buffer size. When K is 0, the M/M/1queue is used. They will be asked as inputs in our program.
   E_N is for the average number of packets in the buffer/queue. P_LOSS is for packet loss probability. P_IDLE is for the proportion of time the server is idle.
   The class of event has attributes of type, time and length, which are used to represent the status of the arrival or departure of a packet.
   We use two lists, event_list and waiting_list for storing arrival/departure events and representing the queues, respectively. The event_list has only one departure event at a time. The waiting list is our server queue.
2) In order to generate the exponential distribution possession process of the inter-arrival time, we used the strategy stated in question 1:

```
while cumulative_arrival_time <= T:
 cumulative_arrival_time += random.exponential(1/lamb)
 length_for_generation = random.exponential(L)
 event_list.append( event(0, cumulative_arrival_time,
length_for_generation) )
```

   Inside the while loop, the random function generates the exponential distribution.
   The process of generating objects of the arrival class will stop at the last generated object as it makes the cumulative arrival time exceed the simulation time, T.
3) In a big while loop, we process packets based on their the type attribute of an event, as introduced in 1).
   For arrival events (event.type as 0), if the server is not free (indicated by the variable flag_busy), we put the event into the M/M/1 queue. And we update the integrated_sum, the variable for tracking the cumulative sum of length, and also the last_change, the variable we used for the time when the number of waiting packets is changed for the last time.

```
    integrated_sum += len(waiting) * (current_event.time - last_change)
    last_change = current_event.time
    waiting.append( current_event )
```

If the server is free, the packet would be added to the event list and the event list would be sorted based on the arrival time of the packet.

```
else:
  #server free, create departure event
  flag_busy = True
  free_time += current_event.time - last_free #update the total free time
  event_list.append( event(1, (current_event.time+(current_event.length/C)), current_event.
  length) )
  event_list.sort( key=lambda x: x.time )
```

For departure events, if the waiting queue is empty, the flag_busy will be set as false so the server is marked as free, and the signal last_free, which records the time when the server is set for free for the last time, would be updated with the end time of the new departure.

```
 #waiting queue is empty, mark server as free
 flag_busy = False
 last_free = current_event.time #set last_free to be the end time of this departure
```

And if the waiting queue is not empty, a packet will be popped from the waiting queue and sent to the event list, with its time attribute accumulated. And the event_list would be re-sorted with the new packet and the integrated_sum would be updated.

```
integrated_sum += len(waiting) * (current_event.time - last_change)
last_change = current_event.time
next_departure = waiting.pop(0)
event_list.append( event(1, (current_event.time+(next_departure.length/C)), next_departure.length) )
event_list.sort( key=lambda x: x.time )
```

-The performance metrics:

In the end of the code, the E_N, P_IDLE and P_LOSS are calculated based on integrated_sum, free_time, number of drop counter and arrival counter, as shown below:

```
E_N = integrated_sum/T
P_IDLE = free_time/T
P_LOSS = drop_counter/arrival_counter
```

Here we use a continuous approach to check the proportion of time the server is idle. So the signal alpha, as suggested, is not used. Here since K = 0, there is no P_LOSS at all, so the P_LOSS is not considered in this scenario.

//

*** I want to described a bit more here about the ‘continuous approach’ which don’t invoke observer events:**

So imagine the time periods with t=[0,4) and t=[4,10], and packets waiting in each time period to be N1=6 and N2=5 respectively. So E[N] in this scenario can be calculated as:

Page 4

E[N] = integrated sum (i.e.Σ length of each time period*the regarding # of packets waiting)/total time elapsed=[(4-0)*6+(10-4)*5]/(4-0)+(10-4)=[24+30]/10=5.4(unit packet)

Right? And you get the idea what we are doing: we divide the whole simulation time into many time periods of shorter length, separated by the points of time at which the # of packets waiting in queue changes. Then we computed the 'integrated sum' - we name it this way as this is similar to integration in Calculus - which equals to:

*Σ length of each time period*the regarding # of packets waiting in that time period*
And then divide the integrated sum by the total simulation time T, and here we get the value of E[N] asked. This method may make the simulator a bit slower, but the benefit is that the result is rather stable even with a comparatively small T.

Also notice that for a packet **BEING** transmitted by the server, we don't consider it as 'in the queue'. E.g. for the M/M/1/K model with K=10, the server can hold up to 11 packets: 1 being transmitted, 10 waiting in the queue. This may cause our E[N] to be approximately 1 smaller than the E[N] gained if we consider a packet being transmitted as 'in the queue'.

//

With L = 2000, C = 1000000, rho = 0.8, T = 100 and K=0, the performance is as the following table:

E[N]	2.8086108037440694
P_IDLE	0.2088565590767134

With L = 2000, C = 1000000, rho = 0.8, T = 200 and K=0, the performance is as the following table:

E[N]	2.927946990680793
P_IDLE	0.20192515150193185

the differences are:
Diff E[N] = (2.927946990680793-2.8086108037440694)/2.8086108037440694= 0.04248939966
Diff P_IDLE = (0.20192515150193185-0.2088565590767134)/0.2088565590767134=

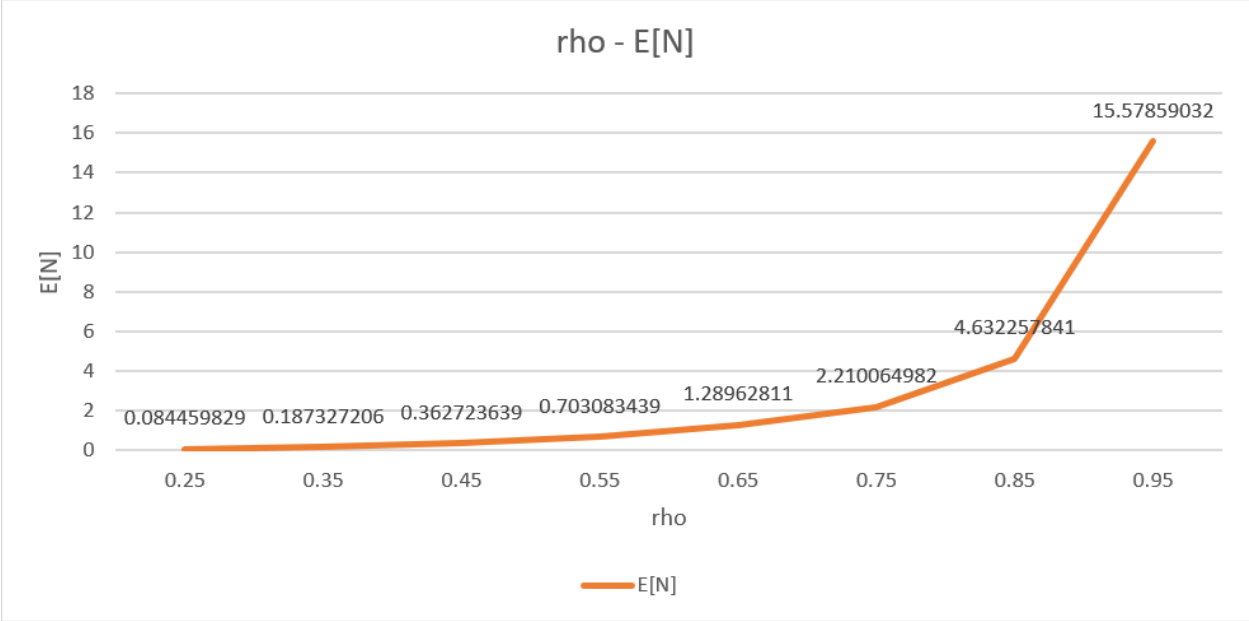
Page 5

−0.03318740673
Both are smaller 5%, which indicates the performance is stable.

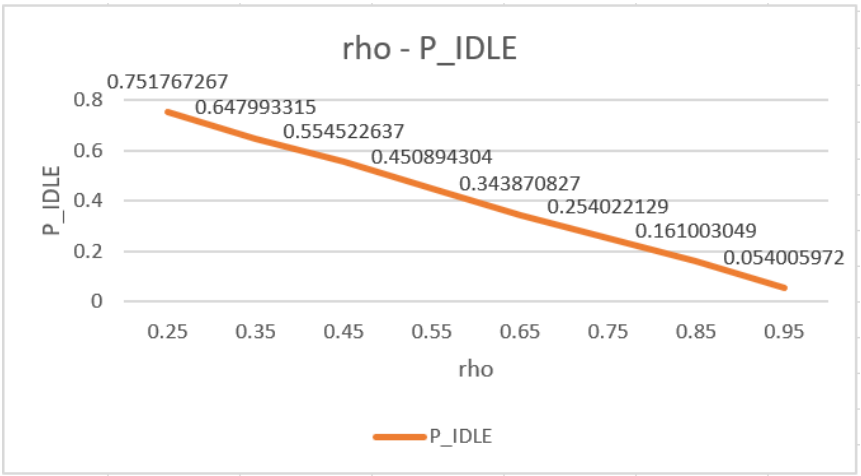
Question 3

With L = 2000, C = 1000000, T = 100, we obtain the following graphs:

- rho - E[N]



- rho - P_IDLE



- Comments
  As shown in both graphs, E[N] is linearly proportional to rho. P_IDLE is inversely proportional to rho.
  This is because lambda = rho*C/L. E[N] has an integrated sum of the time of the packet, or event in our code, which proportionally depends on lambda, so E[N] would increase as rho increases, but not necessarily proportionally.
  P_IDLE is inversely proportional to rho, this is because P_IDLE = free_time/T, where in our code free_time will have to deduct a last_free time. Also, we can understand it as the idle portion of the server decreasing as more packets come in.

Question 4

- With L = 2000, C = 1000000, T = 100, rho = 1.2,
  E[N] = 4984.519298677868, P_IDLE i = 1.462673675002899e-06
- Comments:

The observation is that for the M/M/1 model with infinite queue, if rho>1 then E[N] will be significantly large, while P_idle is almost zero. This is because with rho>1 the server's service capability is actually smaller than the speed at which the packets arrive. Therefore, the server have no time to rest, and the newly arrived packets will pile up in the queue as they are not dropped, and this actually reduces the network's efficiency - actually, the packets arrived after a certain amount of time after the start will never get processed in real life (e.g. imagine a packet that need to theoretically wait one century to be transmitted!). Also, such observation suggests that the M/M/1 model is not quite ideal for real-life scenarios where rho>1, and we may prefer the M/M/1/K model over it in such scenarios though we need to have a trade-off: some packets will be dropped, but at least there won't be 'infinite' waiting or some other problems led by the M/M/1 model.

Question 5

- How to implement the queue:

We use the same waiting queue to implement both M/M/1/K and M/M/1. As described in question 2, K is 0 for M/M/1 queue. Here we use positive K value to simulate M/M/1/K queue. So most of the code is the same for both M/M/1 and M/M/1/K. One difference is in the case of packet arrival. If the waiting queue has space, it will perform the same process as M/M/1 queue, which is described in question 2. If the waiting queue is full, we will just increase the drop_counter by 1. Also, there is an arrival counter, which would be increased by 1 whenever a packet arrives.

```
arrival_counter += 1
if flag_busy:
  #server is not free, check if the waiting queue has free slot(s)
  if K == 0 or len(waiting) < K:
    #there're available slot(s) in the waiting queue
    integrated_sum += len(waiting) * (current_event.time - last_change)
    last_change = current_event.time
    waiting.append( current_event )
  else:
    #the waiting queue is full, dropping packet!
    drop_counter += 1
```

- Performance metrics

For the M/M/1/K queue, an additional P_LOSS is used to track the number of lost packets.

```
E_N = integrated_sum/T
P_IDLE = free_time/T
P_LOSS = drop_counter/arrival_counter
```

With L = 2000, C = 1000000, rho = 0.8, T = 100 and K=10, the performance is as the following table:

E[N]	2.2958438231911056
P_IDLE	0.21891388783701007
P_LOSS	0.01630817167554528

With L = 2000, C = 1000000, rho = 0.8, T = 200 and K=10, the performance is as the following table:

E[N]	2.340538640040775
P_IDLE	0.2134852415859882
P_LOSS	0.019083110321886802

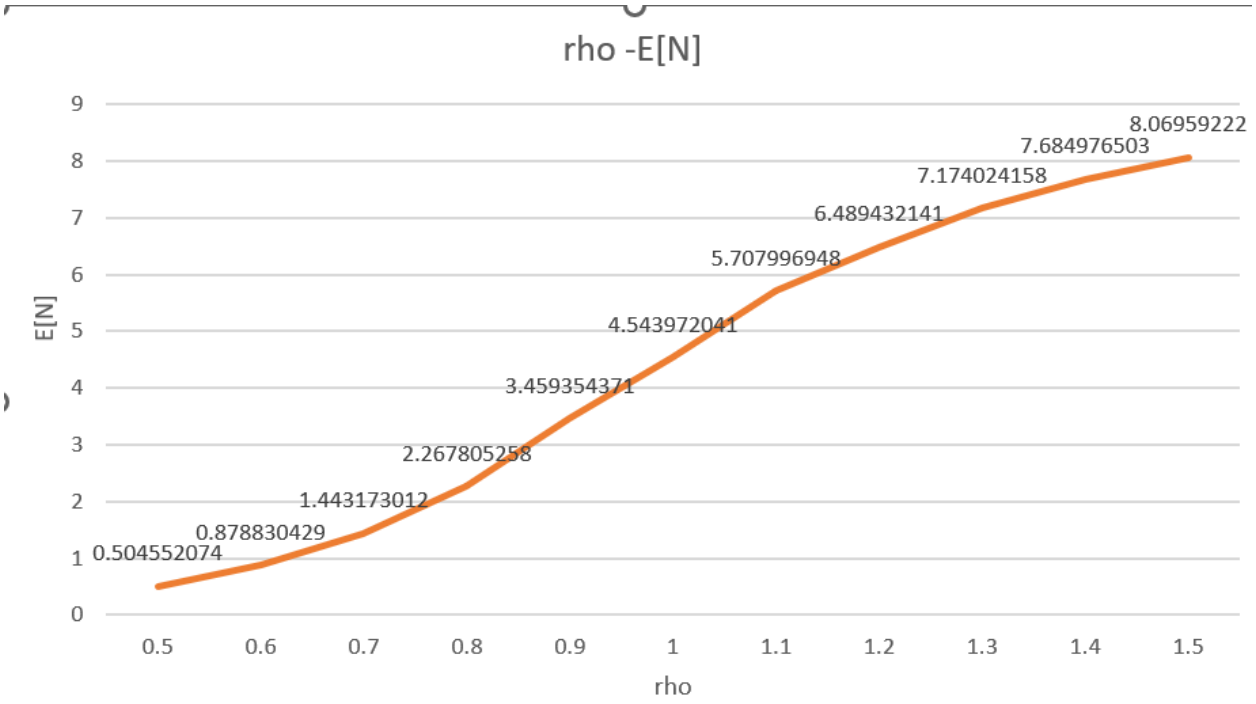
Diff E[N] = (2.340538640040775-2.2958438231911056)/2.2958438231911056=0.01946770786

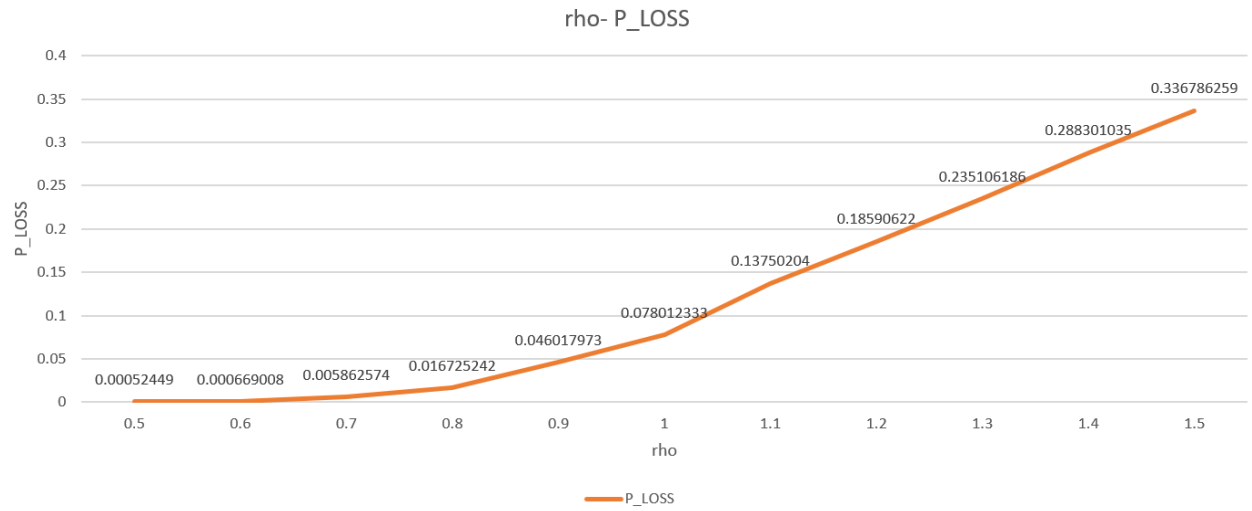
Diff P_LOSS = (0.019083110321886802-0.01630817167554528)/0.01630817167554528=0.1701563303

The difference is within 5%, which indicates the performance is stable.

Question 6

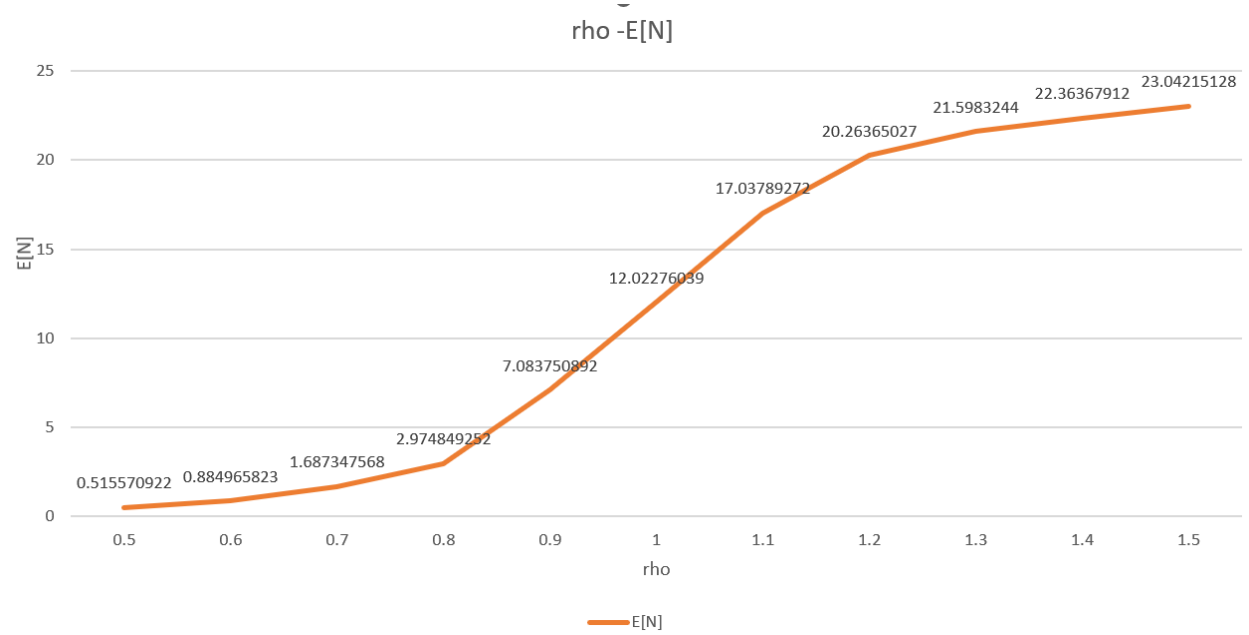
- K=10

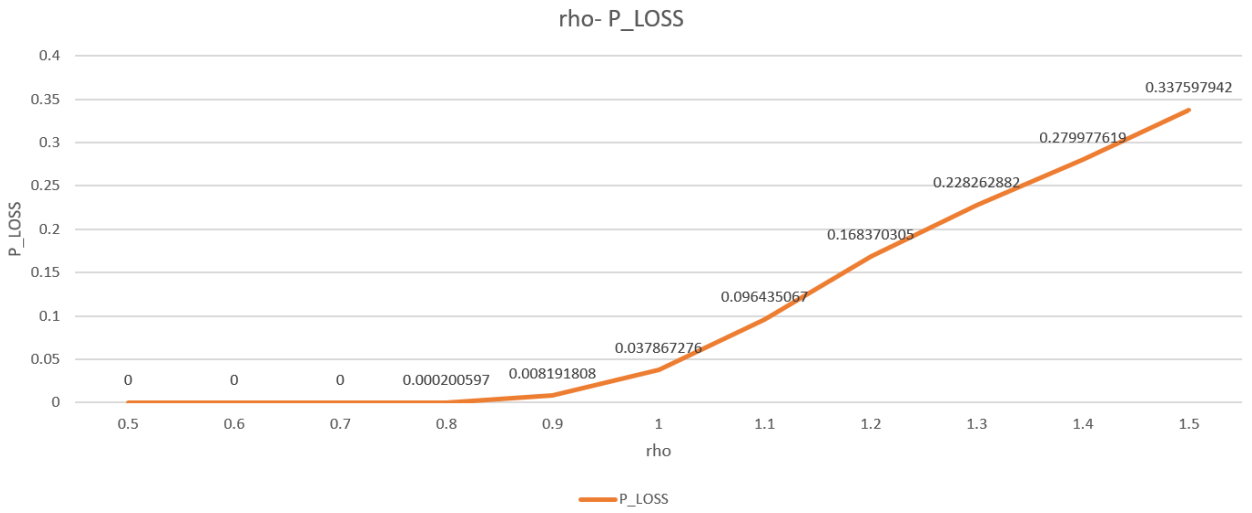




Page 9

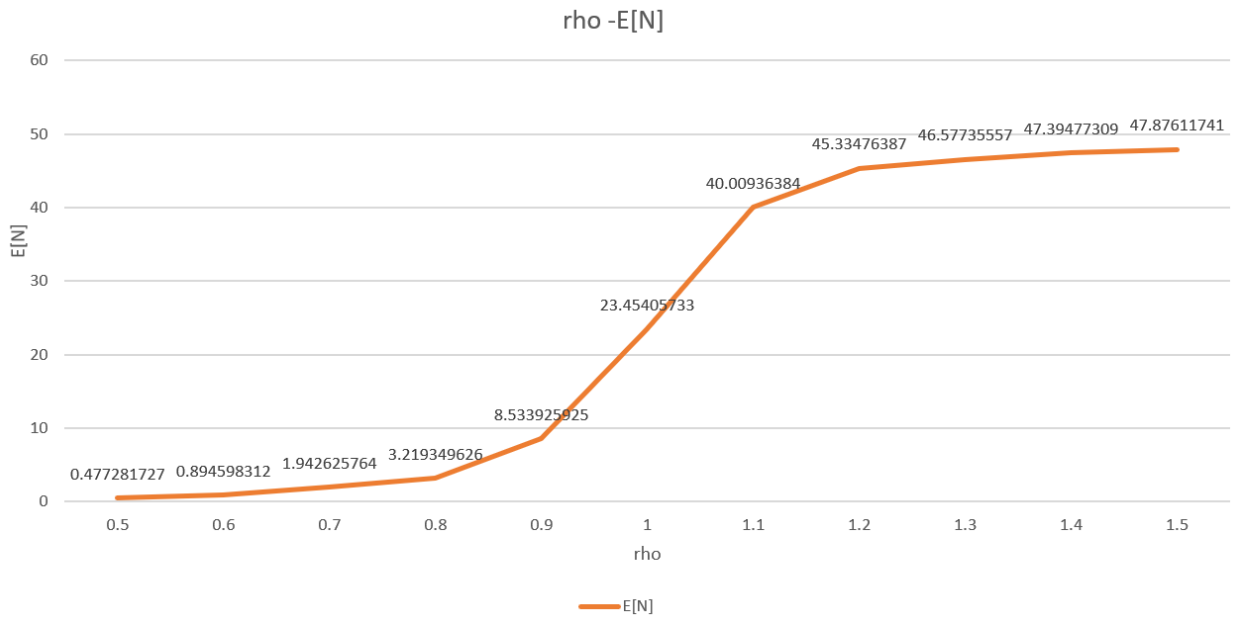
- K=25

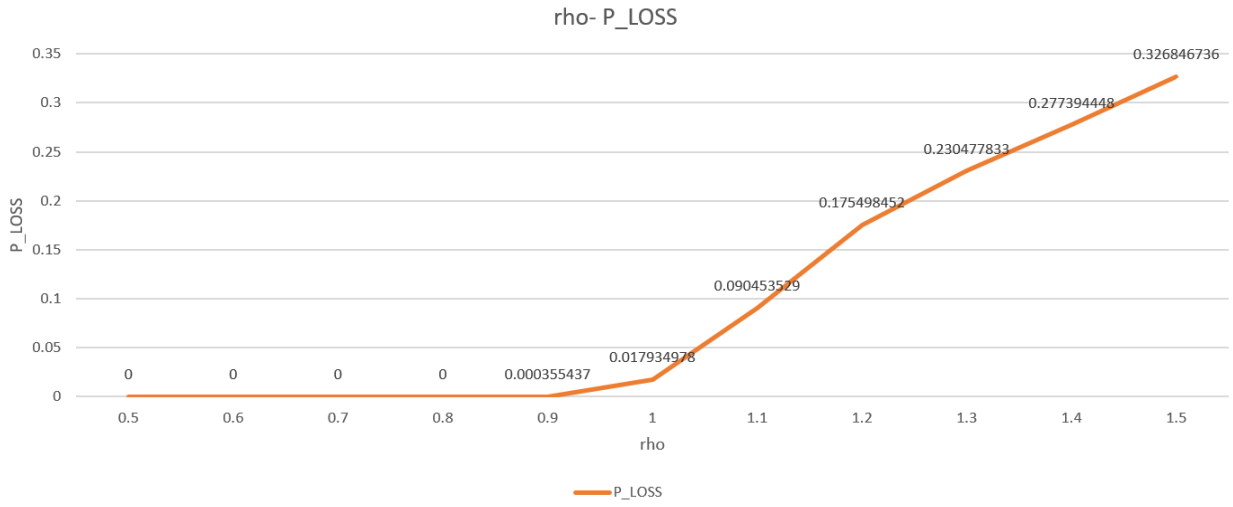




- K=50

Page 10





- Comments
  In the graphs, E[N] always increases as rho increases no matter what the value of K is, this is due to the same reason explained in question 3.
  P_LOSS always increases as rho increases no matter what the value of K is, this is because P_LOSS = drop_counter/arrival_counter, and the number of dropped packets will increase as more packets come in and at a point, the waiting queue will be full. Also, from the P_LOSS graphs of K= 25 and K = 50, we can see that P_LOSS is zero in the beginning, which means the waiting queue is not full and thus no packet is lost. On the graph of K=50, the turning point comes later than the K=25 graph, this is because the K=50 waiting queue has a larger capacity.

Also, the value of E[N] can be larger when the K value increases, which is because rho indicates the quantity of packets coming into the server. When there are more packets, E[N] increases, which explains why E[N] is so large when rho increases.