ECE547 Project1 Ryan Chen

Problem (1)

(a)

Write a computer program that simulates an M/M/1 queue. (IMPORTANT: To "simulate" a queue you should NOT directly use the M/M/1 formula that we derived in the class. Rather, your program should emulate the timing of packet-by-packet arrivals and services in the queue. Then, you should collect the statistics from your simulation in order to answer the following problems. You may want to compare your collected statistics with the M/M/1 formula to verify whether your simulation is correct. Further, you may want to discard the data at the beginning of the simulation because the queue may not have reached steady-state yet.)

```
# Ryan is happy

#simulation variables
temp_res = 1000 # temporal resolution 1000
total_duration = 10 #1
times = []

# arrival variables
_lambda = 8.0
arrival_time = 0
departure_time = 0
arrival_times = [] # arrival_times[i] stores the time at i'th arrival
departure_times = [] # departure_times[i] stores the time at i'th departure
inter_arrival_times = [] # inter_arrival_times[i] stores the time between i'th arrival and (i+1)'th arrival

# service variables
_mu = 10.0
service_time = 0
service_times = []
service_durations = []
busy = 0

# queue variables
queue_length = 0
queue_lengths = []

# simulate queue
n = random.random()
arrival_time = np.random.exponential(1/_lambda)#-math.log(n) / _lambda
arrival_times.append(arrival_time)
inter_arrival_times.append(arrival_time)

#print("time, next arrival time, next service time, queue_length")
for time in range(total_duration*temp_res):
    #print(time, arrival_time*temp_res, service_time*temp_res, queue_length)
    # record state
    times.append(time)
    queue_lengths.append(queue_length)
    # arrival simulation
    if(time > arrival_time*temp_res):
        queue_length += 1
        n = random.random()
        inter_arrival_time = np.random.exponential(1/_lambda)#-math.log(n) / _lambda
        arrival_time = time/temp_res + inter_arrival_time;
        arrival_times.append(arrival_time)
        inter_arrival_times.append(inter_arrival_time)
    # service finish condition
    if(time > service_time*temp_res and queue_length > 0 and busy):
        departure_time = time/temp_res
        departure_times.append(departure_time)
        queue_length -= 1
        busy = 0
    # service start condition
    if(time > service_time*temp_res and queue_length > 0 and not busy):
        n = random.random()
        service_duration = np.random.exponential(1/_mu)#-math.log(n) / _mu
        service_time = time/temp_res + service_duration
        service_times.append(service_time)
        service_durations.append(service_duration)
        busy = 1

```

Here I wrote this simulation to simulate a M/M/1 system.

I keep track of 2 main variables of time, and 2 main variables of state.

1. arrival_time keeps track of the time of next arrival.
2. service_time records the time the server finishes processing the current packet in service.
3. busy indicates the state of the server.
4. queue_length is the state of the queue at every instance and is what we mainly want to know. I record the queue length for each instance in a list queue_lengths .

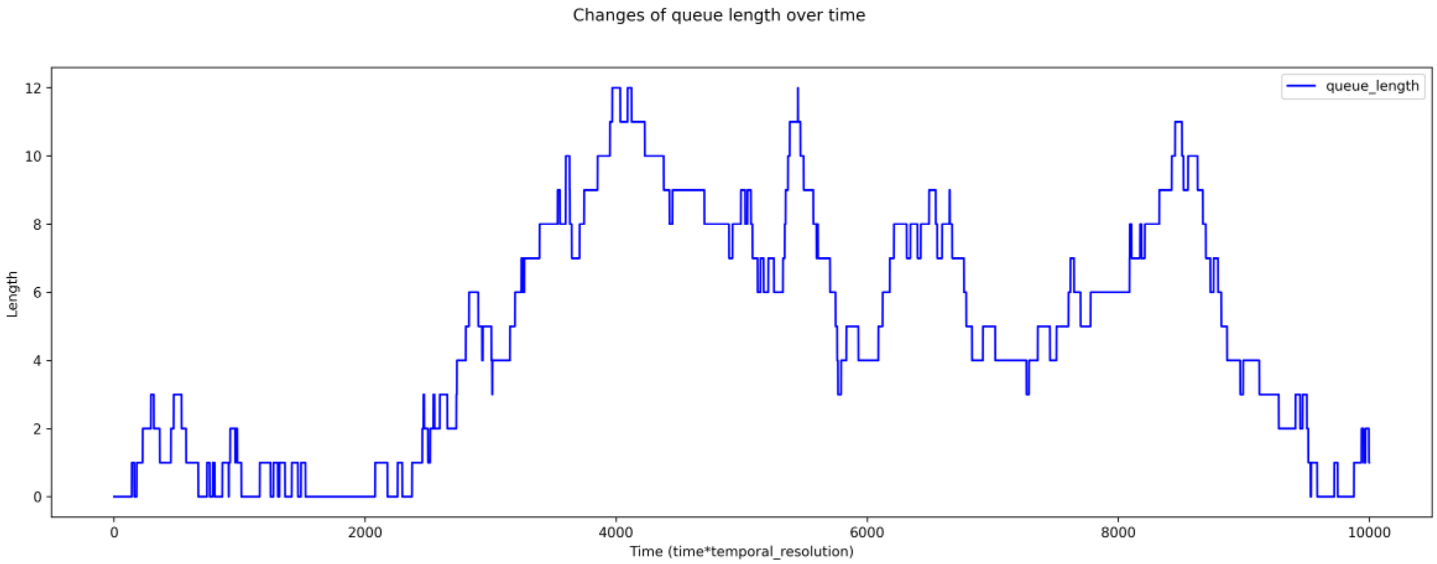
I also have some additional data recorded like the inter_arrival_times, departure_times, and service_durations.

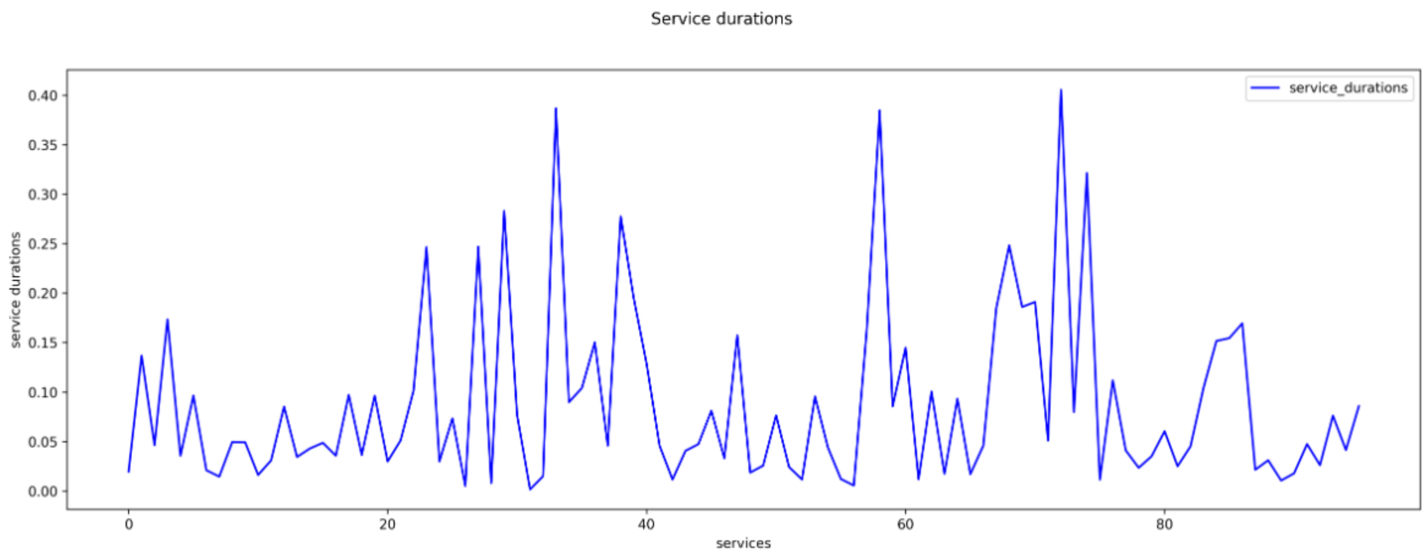
*Note that the temporal resolution temp_res should be sufficiently higher than _lambda + _mu. Since this is the frequency we update our simulation, and _lambda + _mu would be the expected frequency the system changes state.

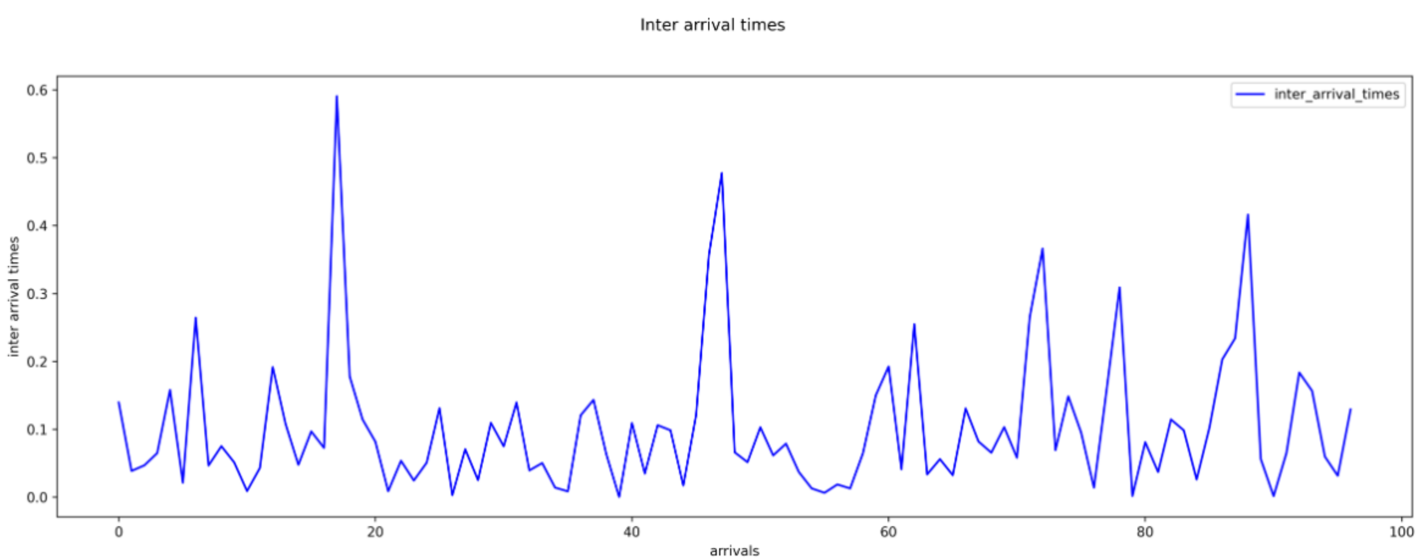
Here, with the help of matplotlib.pyplot we can visualize some data that we collected.

This is with temp_res=1000 (simulate update states every 1ms) and total_duration=10 (sec). For the following analyses of steady state behaviors. We would extend the total_duration to 10000 (secs).

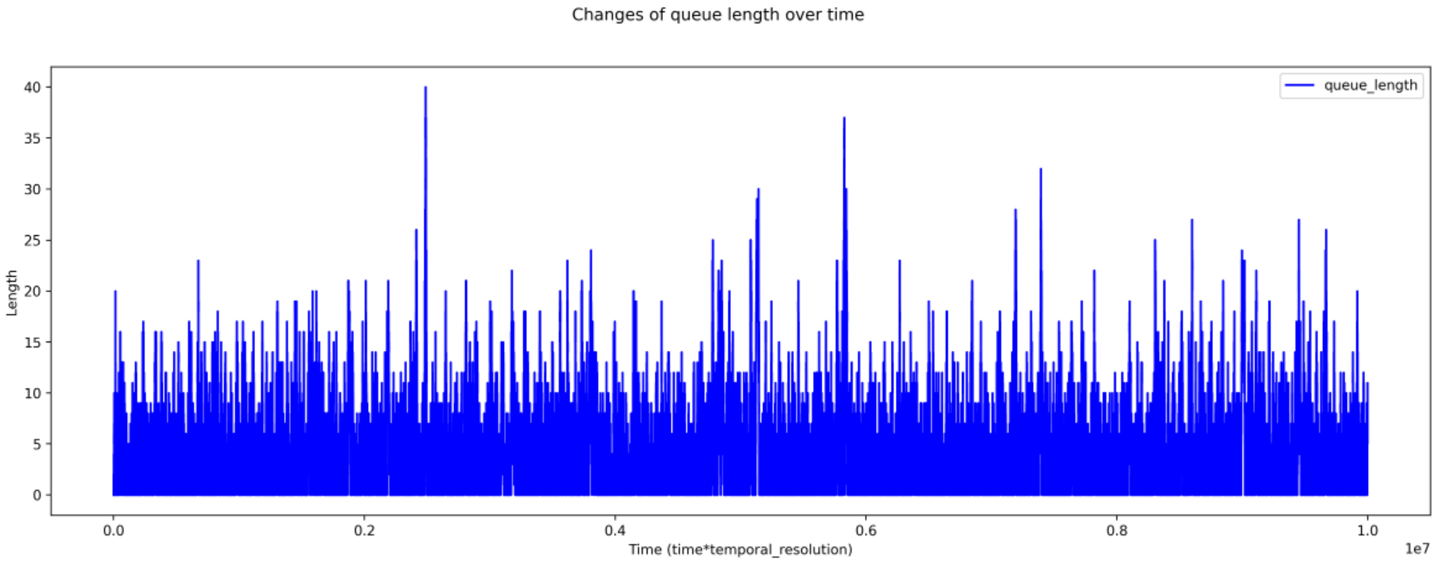
*total_duration=10 here is just for a better view of the changes

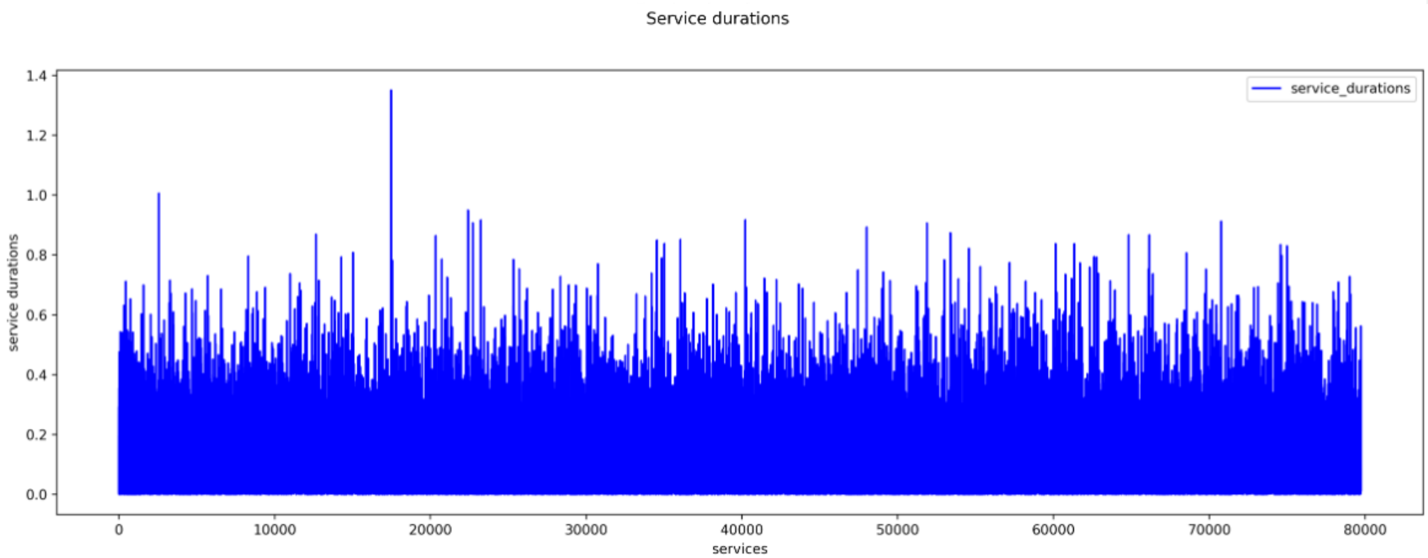


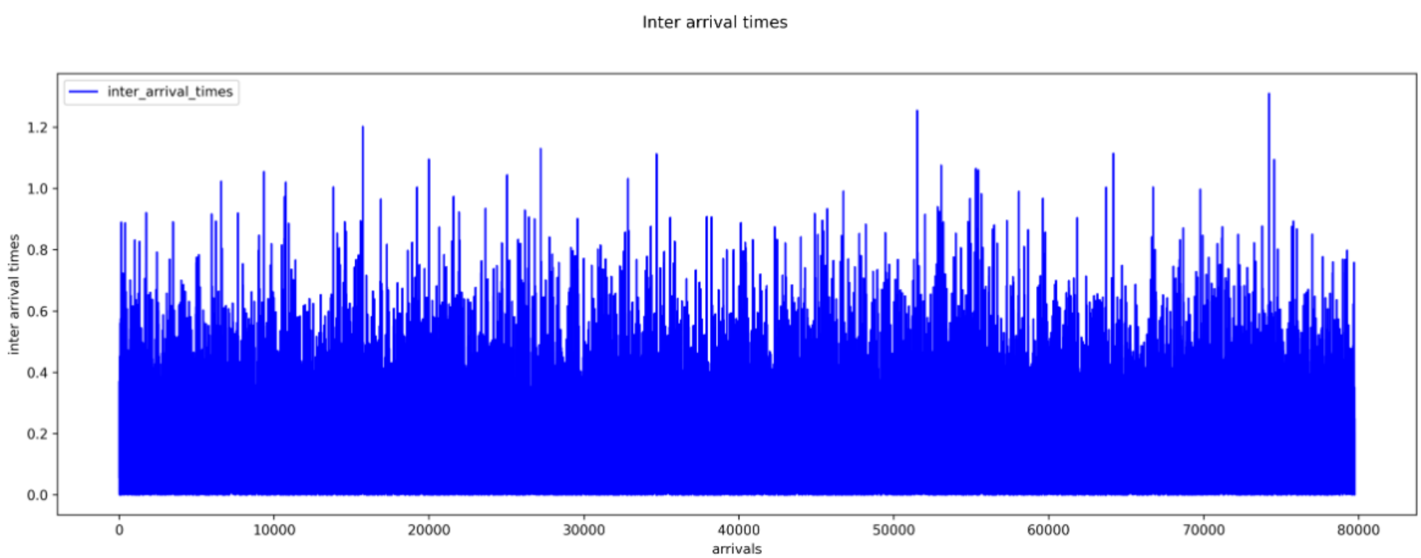




With temp_res=1000, total_duration=10000. We can collect enough data to do our analysis.







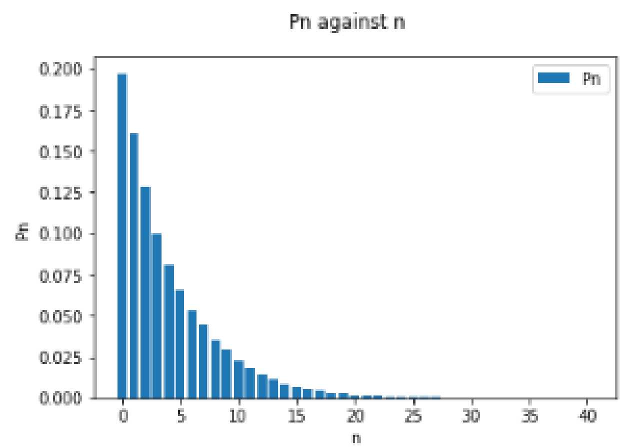
(b)

Based on your simulation, plot Pn against n when λ = 8 and μ = 10.

Pn can be calculated easily from queue_lengths. Which has the state of the queue for every instance in our simulation.

*The print(sum(Pn)) is just to check that Pn adds up to 1. And it did.

```python
# calculate Pn
n = max(queue_lengths)+1
Cn = [0]*n
Pn = []
for data in queue_lengths:
    Cn[data] += 1
print(Cn)
for i in Cn:
    Pn.append(i/len(queue_lengths))
print(Pn)
print(sum(Pn))

# plot Pn against n
services = list(range(len(service_durations)))
fig = plt.figure()
fig.suptitle('Pn against n')
plot = plt.bar(list(range(n)), Pn, label='Pn')
plt.legend(handles=[plot])
plt.xlabel('n')
plt.ylabel('Pn')
plt.show()

```



(c)

From your simulation, find the expected number of packets in your M/M/1 queueing system when ρ = 8/10.

E[n] is just the mean of queue length. And we can ditch some data at the start to minimize initial state discreptancies.

```python
# calculate E[n] with data after the first 10 seconds
print("E[n]: ", statistics.mean(queue_lengths[10*temp_res:]))
```

```
E[n]:  3.9947683683683683
```

(d)

From your simulation, find the expected delay of packets in your M/M/1 queueing system when ρ = 8/10. (IMPORTANT: For part (c) and part (d), you should NOT use Little's Law to derive the queue length or delay. Rather, you should collect them directly from your simulation by averaging the suitable quantities over time or over packets. You may however use Little's Law to verify your results.)

Here, I utilize the fact that queues are FIFO. Which means that the first departure_time is the departure time of the first packet and is linked to the first arrival_time, and so on.

*Note that I use len(departure_times) as index to iterate, since the queue is likely not empty at the end of simulation.

```python
#calculate E[w] with arrival_times and departure_times
delay = []
print("number of arrivals:", len(arrival_times), "\nnumber of departures:", len(departure_times))
for i in range(len(departure_times)):
    delay.append(departure_times[i] - arrival_times[i])
print("\nE[w]: ", statistics.mean(delay))
```

```
number of arrivals: 79745
number of departures: 79734

E[w]:  0.5013609705563243
```

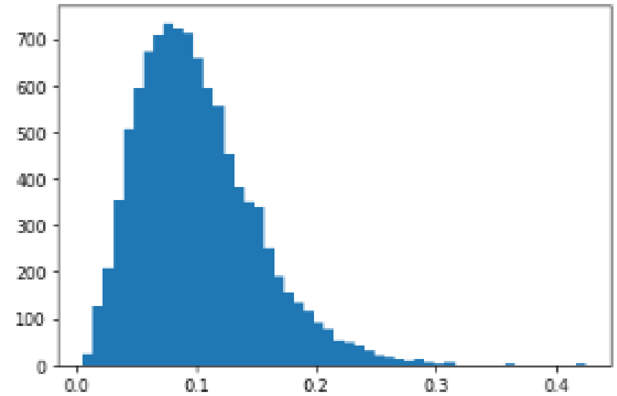
Problem (2)

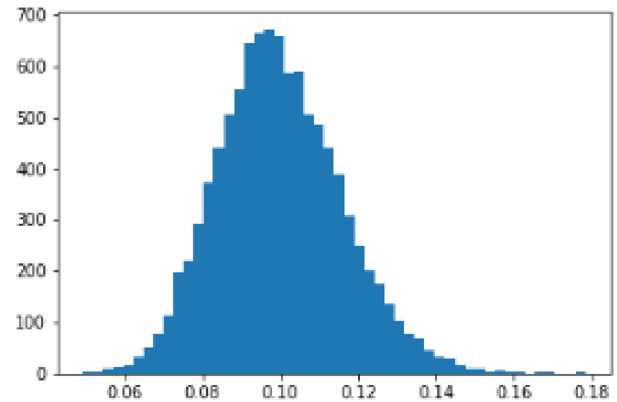
(a)

Write a computer program to generate an Erlang random variable X with k phases, with E[X] = 1/10. An Erlang random variable with k phases is defined as the sum of k i.i.d. exponential random variables. For both the case k = 4 and the case k = 40, use the random numbers generated by your simulation to estimate P(X > x) for each value of x, and plot P(X > x) as a function of x.

First I wrote some code to simulate the erlang variable, and plot to make sure it look right.

```
import random
import numpy as np
import statistics
import matplotlib.pyplot as plt

# Ryan is happy

def erlang(e, k, n=1):
    ret = []
    _lambda = k/e
    for num in range(n):
        x = 0
        for i in range(k):
            x += (-np.log(random.random()) / _lambda)
        ret.append(x)
    return ret

def exponential(_lambda, n):
    ret = []
    for num in range(n):
        ret.append(-np.log(random.random()) / _lambda)
    return ret

erl_k4 = erlang(1/10, 4, 10000)
erl_k40 = erlang(1/10, 40, 10000)

plt.hist(erl_k4, bins=50)
plt.show()

plt.hist(erl_k40, bins=50)
plt.show()
print(statistics.mean(erl_k4), statistics.mean(erl_k40))
```





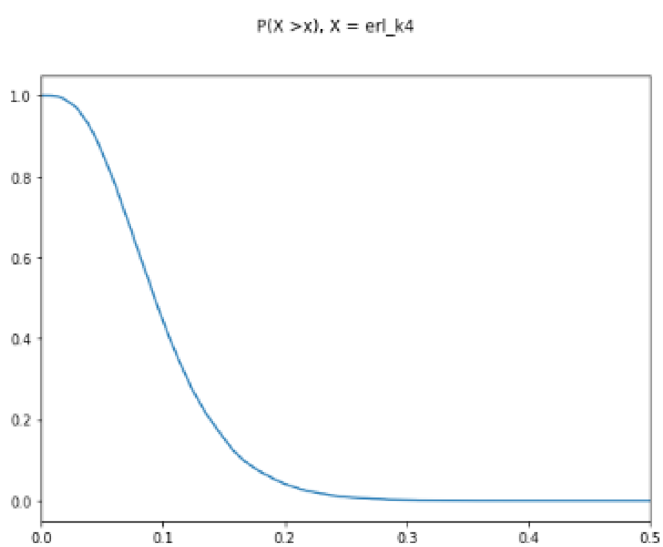

```
0.1005785444230746 0.09982178646809586
```

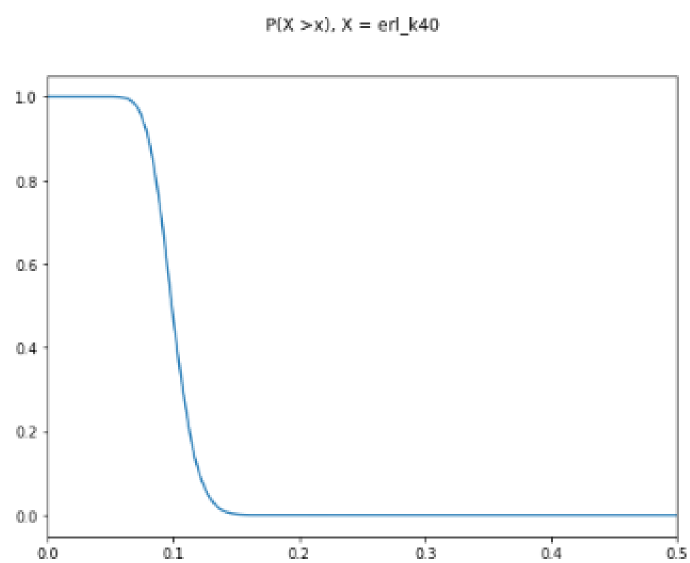
To plot P(X>x), I came up with this method.

1. Have the list of erlang random variables in reversed sorted order. [largest, … , smallest]
2. View the indexes as "How many points in the list has larger value then it" (X>x)
3. Normalize data to [0,1]

*A small tweak is to append a datapoint with value 0 at the end of the reverse sorted list, and a datapoint with value that would normalize to 1 at the start of the reverse sorted list. Else the plot may be missing in [0, value of the smallest point], and [value of the largest point, infinity]

```
# efficient way of plotting P(X > x)
_erl_k4 = erl_k4
_erl_k4.sort(reverse=True) # have _erl_k4 in sorted descending order so the index of i would be #of points with val greater than i
_erl_k4.insert(0,len(_erl_k4)) # insert at front a point that would show on the plot as (0,1)
_erl_k4.append(0) # append at end a point that would show on the plot as (x,0), x being the largest x-axis value in the plot

plt.figure(figsize=(8,6)).suptitle('P(X >x), X = erl_k4')
plt.plot(_erl_k4, np.linspace(0, 1, len(_erl_k4)))
plt.xlim([0, 0.5])
plt.show()

_erl_k40 = erl_k40
_erl_k40.sort(reverse=True)
_erl_k40.insert(0,len(_erl_k40))
_erl_k40.append(0)

plt.figure(figsize=(8,6)).suptitle('P(X >x), X = erl_k40')
plt.plot(_erl_k40, np.linspace(0, 1, len(_erl_k40)))
plt.xlim([0, 0.5])
plt.show()
```





(b)

Write a computer program that simulates an M/Ek/1 queue. Here, Ek is an Erlang random variable with k phase.

```python
def erlang_single_val(e, k):
    _lambda = k/e
    x = 0
    for i in range(k):
        x += (-np.log(random.random()) / _lambda)
    return x

def simulate_queue(k=4, _lambda=8, _mu=10, temp_res=1000, total_duration=10000):
    #simulation variables
    # temp_res = 1000 # temporal resolution 1000
    # total_duration = 10000 #10000
    global times
    times = []

    # arrival variables
    arrival_time = 0
    departure_time = 0
    arrival_times = [] # arrival_times[i] stores the time at i'th arrival
    departure_times = [] # departure_times[i] stores the time at i'th departure
    global inter_arrival_times
    # inter_arrival_times[i] stores the time between i'th arrival and (i+1)'th arrival
    inter_arrival_times = []

    # service variables
    e = 1/_mu
    service_time = 0.0
    service_times = []
    global service_durations
    service_durations = []
    busy = 0

    # queue variables
    queue_length = 0
    global queue_lengths
    queue_lengths = []


    n = random.random()
    arrival_time = np.random.exponential(1/_lambda)#-math.log(n) / _lambda
    arrival_times.append(arrival_time)
    inter_arrival_times.append(arrival_time)

    #print("time, next arrival time, next service time, queue_length")
    for time in range(total_duration*temp_res):
        #print(time, arrival_time*temp_res, service_time*temp_res, queue_length)
        # record state
        times.append(time)
        queue_lengths.append(queue_length)
        # arrival simulation
        if(time > arrival_time*temp_res):
            queue_length += 1
            inter_arrival_time = np.random.exponential(1/_lambda)
            arrival_time = time/temp_res + inter_arrival_time;
            arrival_times.append(arrival_time)
            inter_arrival_times.append(inter_arrival_time)
        # service finish condition
        if(time > service_time*temp_res and queue_length > 0 and busy):
            departure_time = time/temp_res
            departure_times.append(departure_time)
            queue_length -= 1
            busy = 0
        # service start condition
        if(time > service_time*temp_res and queue_length > 0 and not busy):
            service_duration = erlang_single_val(e, k)
            service_time = time/temp_res + service_duration
            service_times.append(service_time)
            service_durations.append(service_duration)
            busy = 1

simulate_queue(4, 8, 10)
```

For this part, I modified my code for simulation from Problem 1. And also made it into a function for convenience.

The main variables is unchanged. Only the random process for service_duration changed from poisson to erlang.

The erlang_single_val(e, k) function returns a erlang random variable with E[X] = e and k phases.

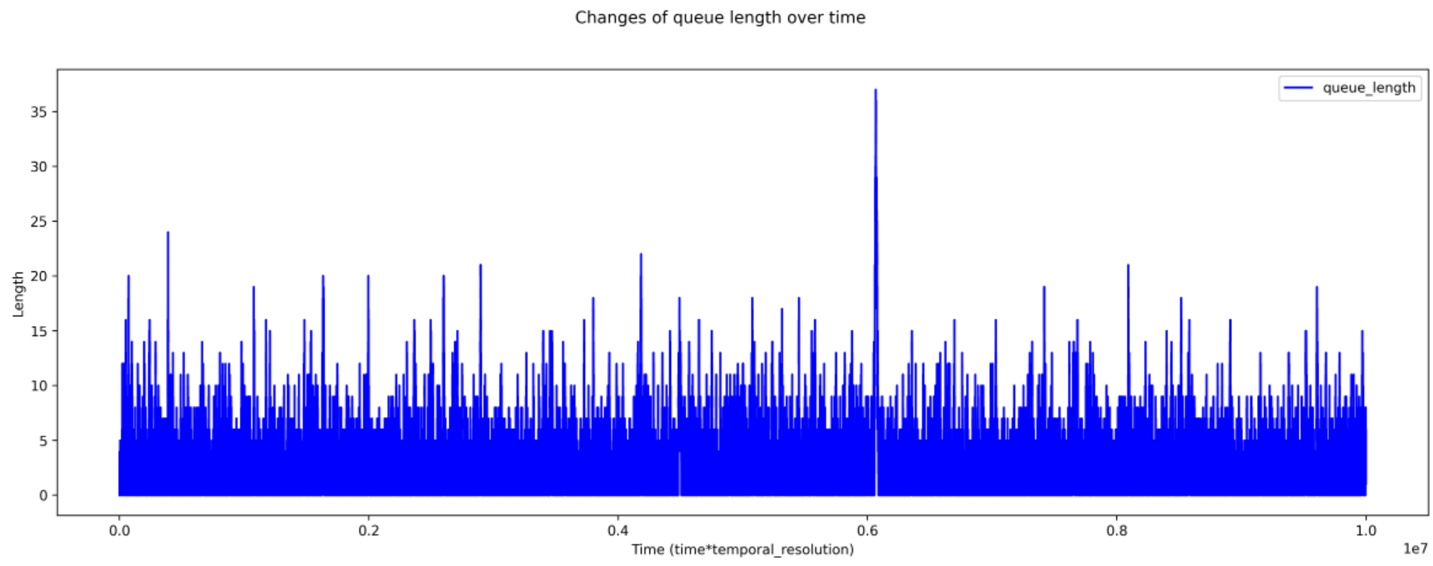
```python
simulate_queue(4, 8, 10)
print("E[n]: ", statistics.mean(queue_lengths))
# print(len(queue_lengths))
```

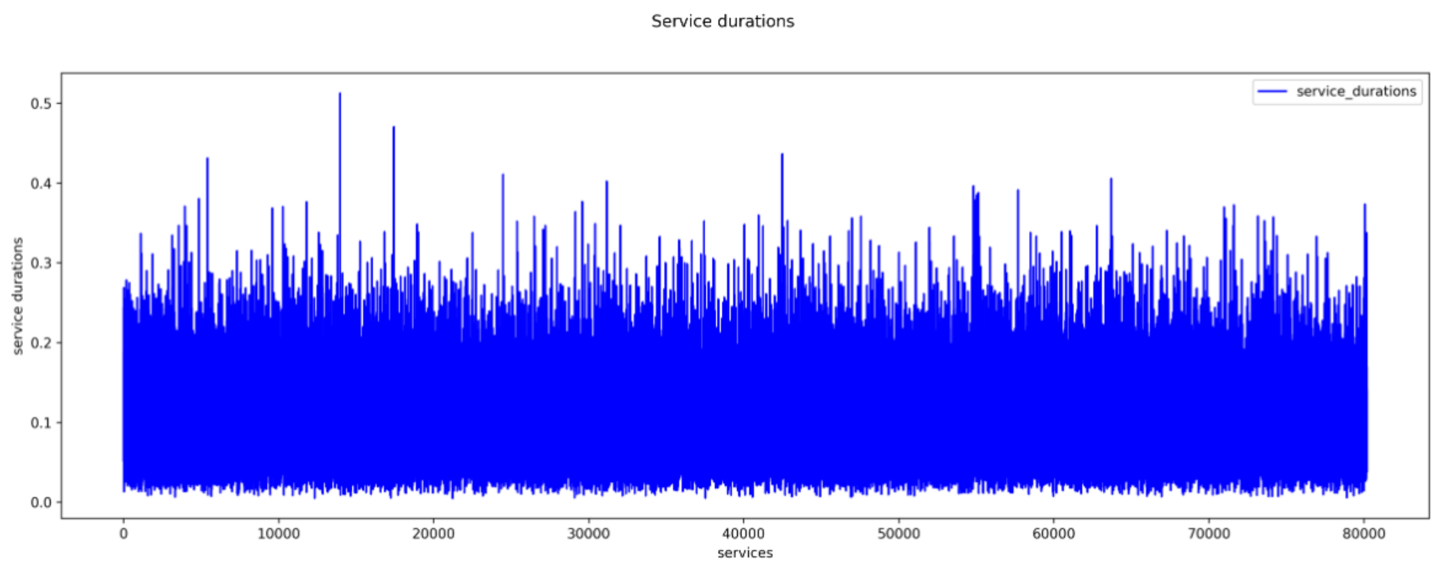
```
--INSERT--
E[n]:  2.831378
```

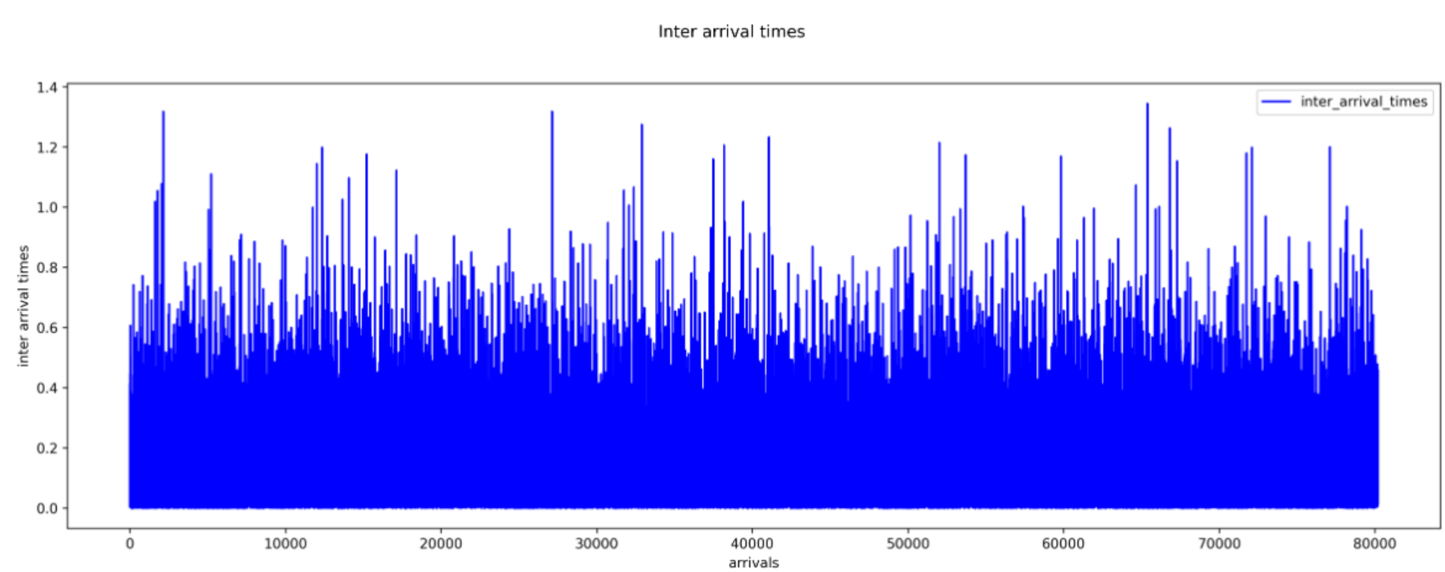
In addition, I also checked for accuracy with E[n] and it seems accurate.

Theoretical E[n] is 2.8

The same plotting code generates theses visualizations.







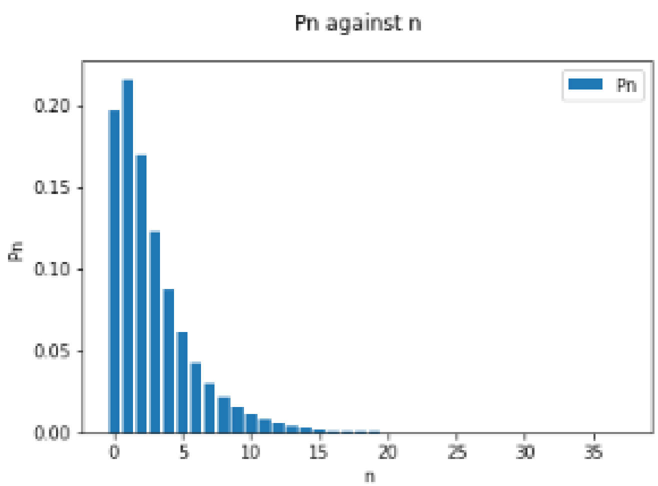
(c)

Based on your simulaton, plot Pn against n when k = 4, λ = 8 and μ = 10. Note that μ = 10 implies that the expected service time is 1/μ = 1/10. Also, find the expected number of packets in the system. How do these results compare with your M/M/1 results in (1)?

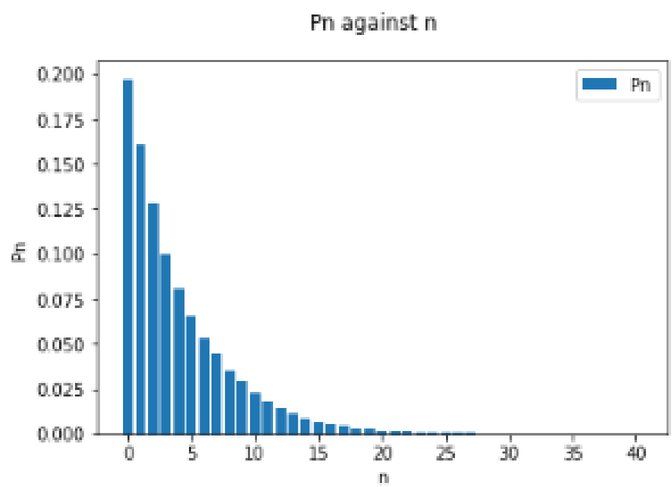
I also reused my code for plotting Pn here.

On the left is Pn against n for M/Ek/1, on the right is for M/M/1.

I feel like the distribution is more concentrated in M/Ek/1 case., also the highest bin is shifted to the second bin compared to M/M/1.



Pn against n, M/Ek/1



Pn against n, M/M/1

(d)

Now use k = 40. Vary the utilization ρ of the M/Ek/1 queue and run your simulations again over a range of ρ. From your simulation, find the expected number of packets in the system at each utilization level ρ. Plot the expected number of packets in the system against the utilization level ρ when k = 40. Also, plot the expected number of packets in the system against ρ for an M/D/1 queue using the theoretical results in class, and compare the results with your simulation. What does this tell you and why?

With simulate_queue() function. I ran simulation with ρ (rho) = 0.1 ~ 1 with steps 0.1.

Each result is plotted and appended at the end.

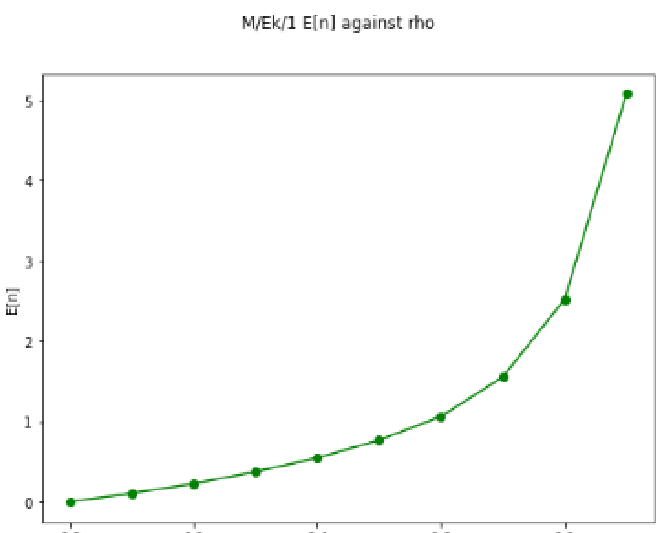
Then I plotted all the E[n]s against ρ. Surprisingly, they look the almost the same.

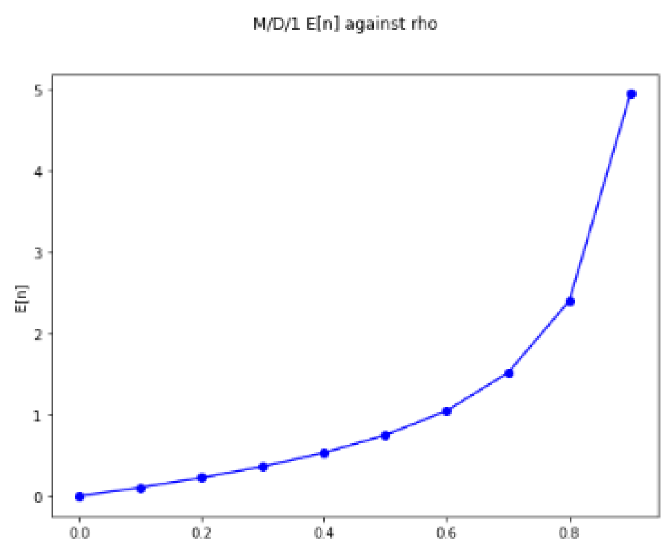
From Problem 2 (a), we can notice that as k increases in an Ek system, the P(X>x) becomes closer to a step.

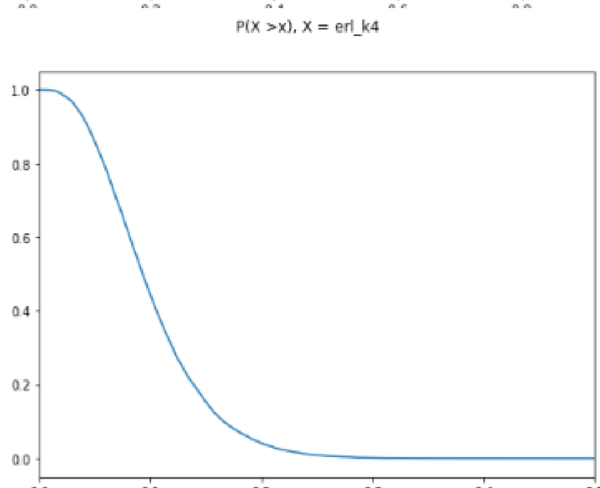
This indicates that P(x) concentrates close to E[X] as k increases. i.e. P(X close to E[X]) increases.

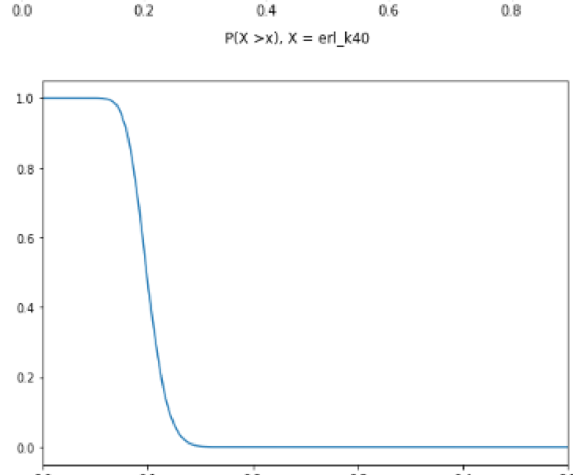
For M/D/1 systems, P(X=E[X]) is basically 1. Therefore as k increases in M/Ek/1 systems, it behaves more and more like a M/D/1 system.

```
M/Ek/1 Ens:  [0, 0.1053168, 0.2242908, 0.372851, 0.5450714, 0.765561, 1.0624724, 1.5520248, 2.5209337, 5.0789495]
M/D/1 Ens:  [0.0, 0.10555555555555557, 0.225, 0.3642857142857143, 0.5333333333333334, 0.75, 1.0500000000000003, 1.516666666666667, 2.400000000000001, 4.950000000000002]
```









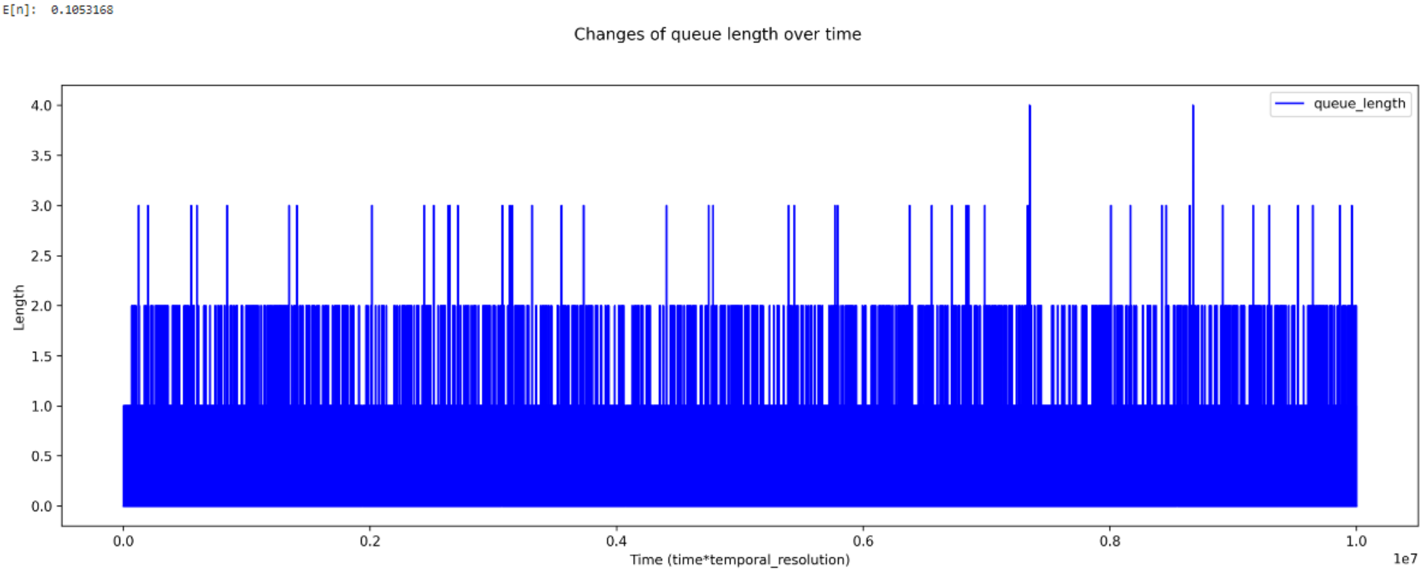
Here are all the result of the simulations. E[n]s are saved in En1, En2, En3…

```
Ens = [0, En1, En2, En3, En4, En5, En6, En7, En8, En9]
print("M/Ek/1 Ens: ", Ens)
plt.figure(figsize=(8,6)).suptitle('M/Ek/1 E[n] against rho')
plt.plot(np.linspace(0.0, 0.9, 10),Ens, 'go-')
plt.xlabel('rho')
plt.ylabel('E[n]')
Ens_MD1 = [i+1/2*(i*i)/(1-i) for i in np.linspace(0.0, 0.9, 10)]
print("M/D/1 Ens: ", Ens_MD1)
plt.figure(figsize=(8,6)).suptitle('M/D/1 E[n] against rho')
plt.plot(np.linspace(0.0, 0.9, 10),Ens_MD1, 'bo-')
plt.xlabel('rho')
plt.ylabel('E[n]')
plt.show()
```

```
M/Ek/1 Ens:  [0, 0.1053168, 0.2242908, 0.372851, 0.5450714, 0.765561, 1.0624724, 1.5520248, 2.5209337, 5.0789495]
M/D/1 Ens:  [0.0, 0.10555555555555557, 0.225, 0.3642857142857143, 0.5333333333333334, 0.75, 1.0500000000000003, 1.516666666666667, 2.400000000000001, 4.950000000000002]
```

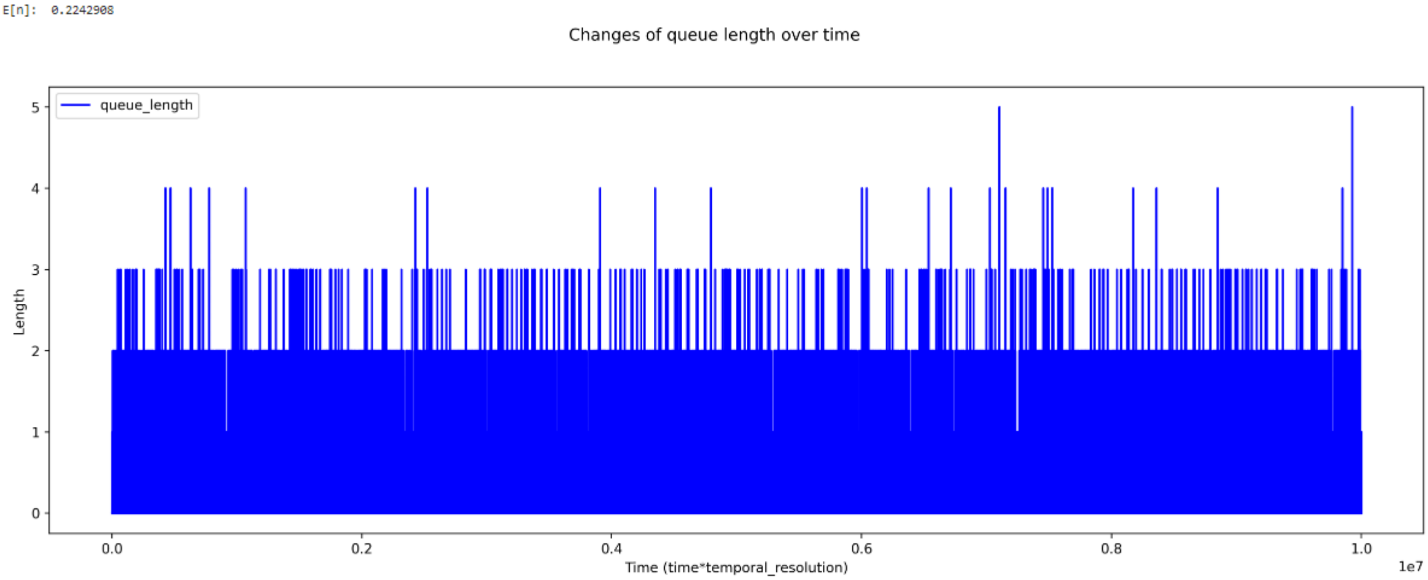
```
simulate_queue(40, 1, 10) # rho = 0.1
En1 = statistics.mean(queue_lengths)
print("E[n]: ", En1)
plot_queue_data()
```

```
E[n]:  0.1053168
```



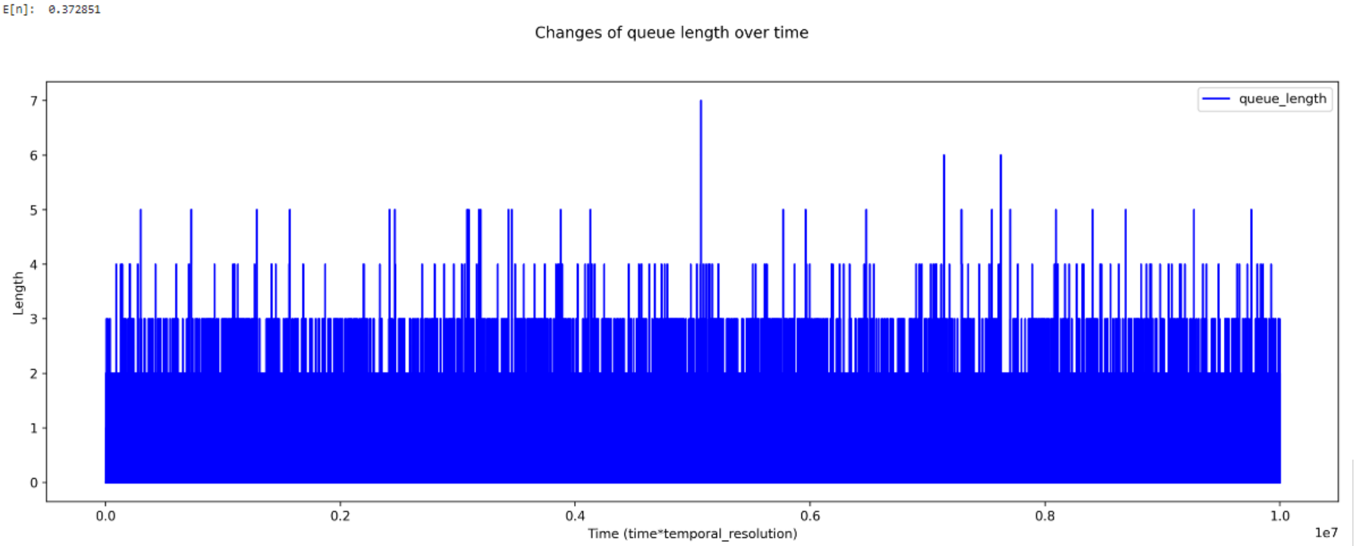

```
simulate_queue(40, 2, 10) # rho = 0.2
En2 = statistics.mean(queue_lengths)
print("E[n]: ", En2)
plot_queue_data()
```

```
E[n]:  0.2242908
```



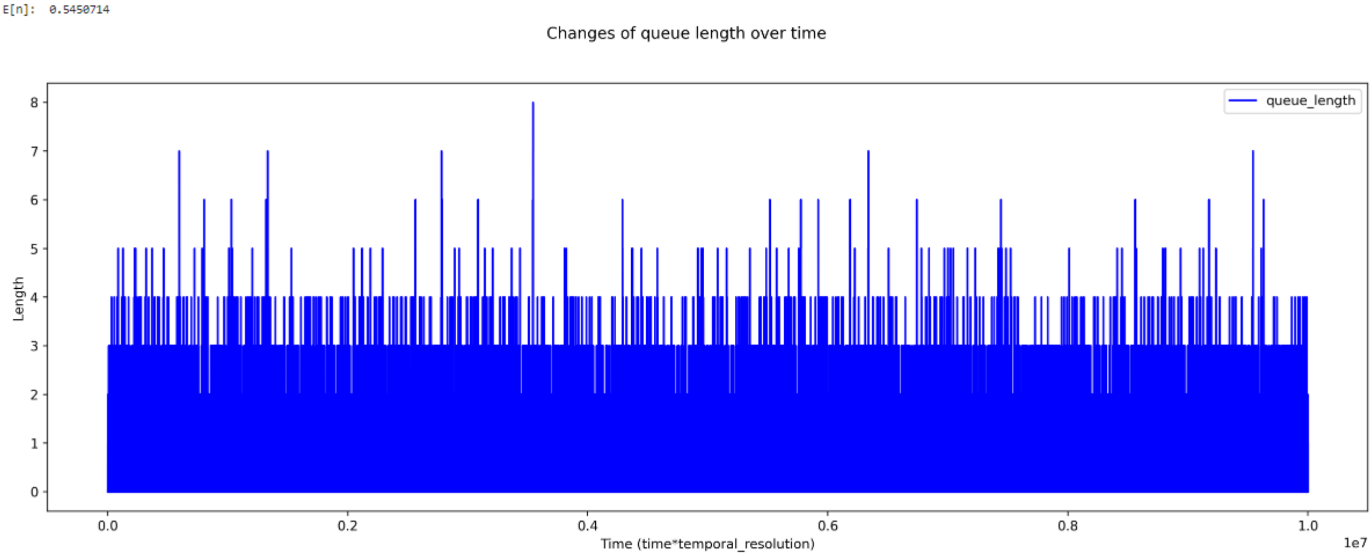
```
simulate_queue(40, 3, 10) # rho = 0.3
En3 = statistics.mean(queue_lengths)
print("E[n]: ", En3)
plot_queue_data()
```

```
E[n]:  0.372851
```



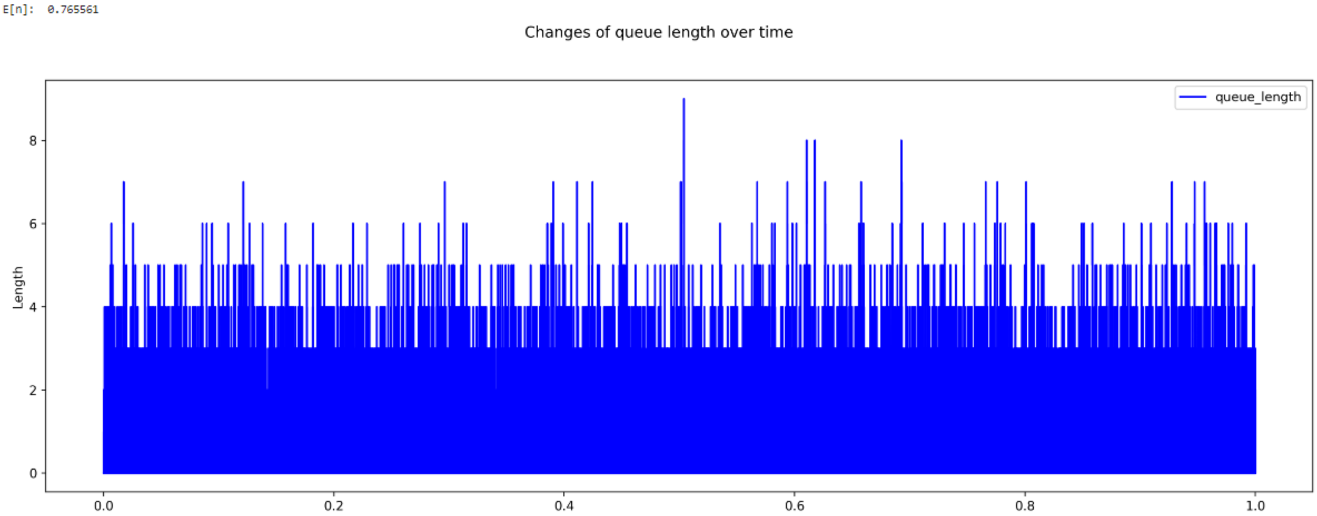

```
simulate_queue(40, 4, 10) # rho = 0.4
En4 = statistics.mean(queue_lengths)
print("E[n]: ", En4)
plot_queue_data()
```

```
E[n]:  0.5450714
```




```
simulate_queue(40, 5, 10) # rho = 0.5
En5 = statistics.mean(queue_lengths)
print("E[n]: ", En5)
plot_queue_data()
```

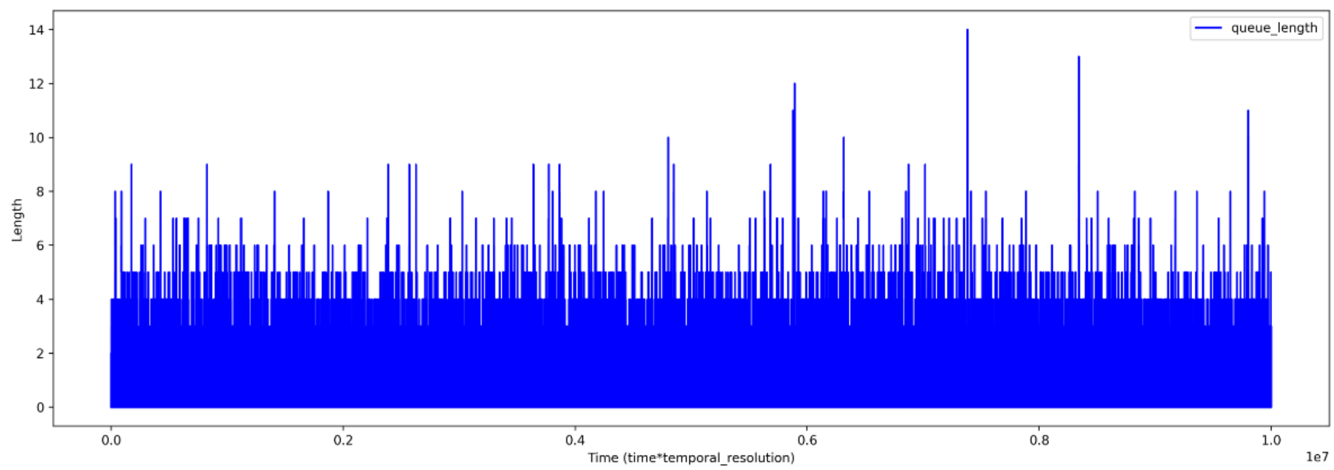
```
E[n]:  0.765561
```



```
simulate_queue(40, 6, 10) # rho = 0.6
En6 = statistics.mean(queue_lengths)
print("E[n]: ", En6)
plot_queue_data()
```

E[n]: 1.0624724

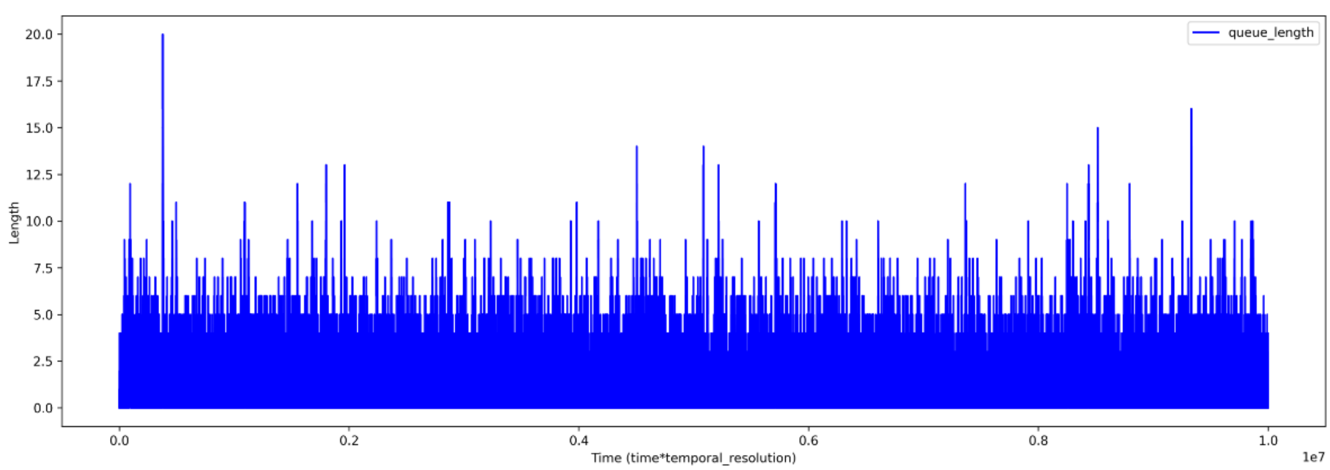
Changes of queue length over time




```
simulate_queue(40, 7, 10) # rho = 0.7
En7 = statistics.mean(queue_lengths)
print("E[n]: ", En7)
plot_queue_data()
```

E[n]: 1.5520248

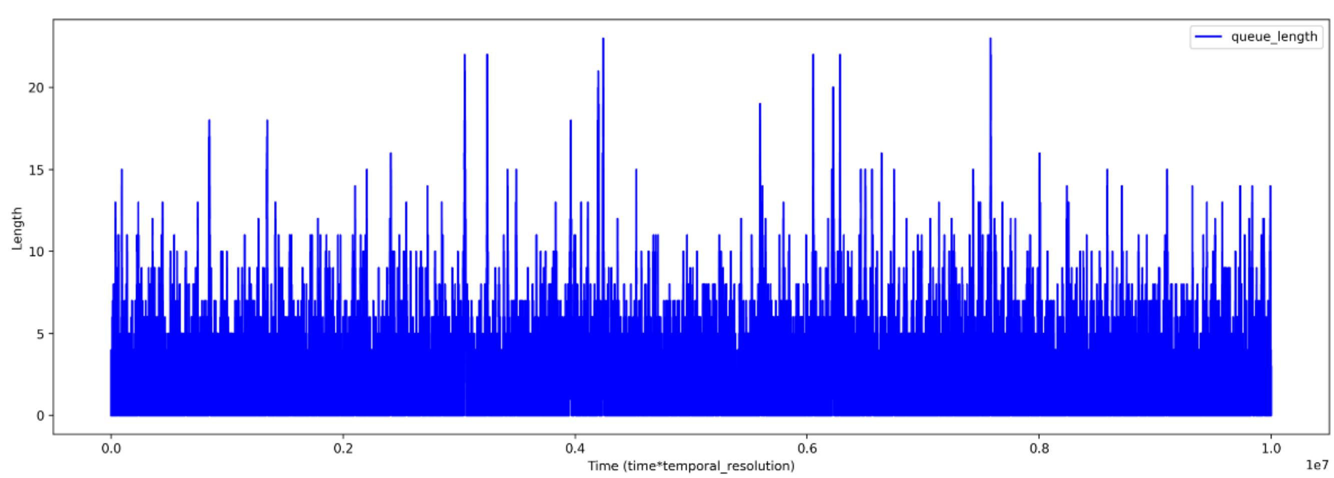
Changes of queue length over time




```
simulate_queue(40, 8, 10) # rho = 0.8
En8 = statistics.mean(queue_lengths)
print("E[n]: ", En8)
plot_queue_data()
```

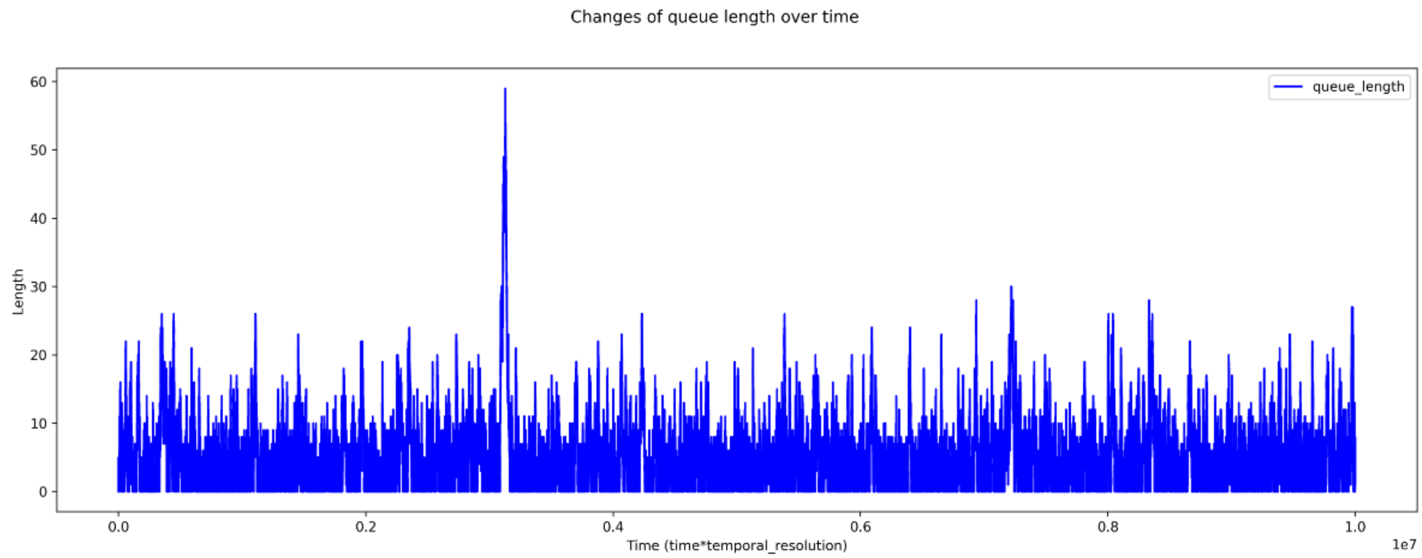
E[n]: 2.5209337

Changes of queue length over time



```
simulate_queue(40, 9, 10) # rho = 0.9
En9 = statistics.mean(queue_lengths)
print("E[n]: ", En9)
plot_queue_data()
```

```
E[n]:  5.0789495
```




```
simulate_queue(40, 10, 10) # rho = 1, we won't use this since it diverges
En10 = statistics.mean(queue_lengths)
print("E[n]: ", En10)
plot_queue_data()

```

```
E[n]:  742.3818324
```