Student name: Steven Xie

Student number: 998979627

Program & Department: MEng MIE

Homework 2

Problem 1.

$Y_1 = 0$, $Y_2 = \max(0, X_1 - A_2)$, $Y_i = \max(0, Y_{i-1} + X_{i-1} - A_i)$, i = 1, 2, …10

So $Y_{10} = \max(0, Y_9 + X_9 - A_{10})$, with n replication the $E(Y_{10}) = \frac{1}{n}\sum_1^n Y_{10}$. The estimator is not biased since Y_{10} in each replication is independent and unbiased.

The goal is to estimate the expected waiting time of the 10^{th} customer to enter the system. We use n = 1000 replications to generate 1000 Y_{10} and compute the sample average. The warmup period should not be included in the model since we are not trying to reach the steady state and compute the mean of waiting time. The interarrival time and service time are unchanged.

The estimate of Y_{10} is $E(Y_{10}) = \frac{1}{n}\sum_1^n Y_{10} = 1.28$, with a 95% CI of [1.19, 1.36]

Code shown below.

```python
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

# define the function to compute the mean and confidence interval
def t_mean_confidence_interval(data, alpha):
    a = 1.0 * np.array(data)
    n = len(a)
    m, se = np.mean(a), np.std(a, ddof=1)
    h = stats.t.ppf(1 - alpha / 2, n - 1) * se / np.sqrt(n)
    return m, "+/-", h


LastWait = []
print("Rep", "10th Customer Wait Time")

n = 1000 # set the replication time to 1000
d = 10 # set the # of customer arrival to 10
MeanTBA = 1.0  # average interarrival time
MeanST = 0.8  # average service time

np.random.seed(1)

for Rep in range(0, n):
    Y = 0
    ListY = []
    for i in range(0, d, 1):
        A = np.random.exponential(MeanTBA, 1) # set the arrival time
        X = np.sum(np.random.exponential(MeanST / 3, 3)) # set the service time
        Y = max(0, Y + X - A) # compute the wait time of customer i
        ListY.append(float(Y)) # record the wait time of all customers in a list
    LastWait.append(ListY[-1]) # add the last customer's wait time in the list
    # print the last customer's waiting time for each rep
    print(Rep + 1,'    ', ListY[-1])

print("95% CI for the 10th customer's wait:",
      t_mean_confidence_interval(LastWait, 0.05))
```

Result shown below

```
Rep 10th Customer Wait Time
1     0.0
2     0.0
3     0.0
4     1.6127778724891926
5     2.0346840655617395
6     1.6939625965501701
7     0.9158242869262436
8     0.5040490730201341
9     1.253795029294279
10     2.5602790895239718
11     0.0
12     0.014243074687452673
13     1.557763729395523
14     0.02587234721266174
15     0.011017489215254
16     0.0
17     2.284294795694494
18     0.6389890172303587
19     0.24337697875384734
20     2.930491163469544
21     0.8998786498149766
```

Rep 22 – 975 in between, not shown for space saving.

```
976     3.079819897754689
977     0.09281076557747803
978     3.1174685405659064
979     4.605628037953515
980     1.4088210265389218
981     0.0
982     0.46511138388176665
983     1.2974409942805152
984     0.0
985     0.0
986     0.0
987     0.0
988     1.61475438430176
989     2.198573141702414
990     0.0
991     0.0
992     2.7700117022815895
993     0.0
994     2.7507140022331704
995     1.7534116129845003
996     0.0
997     1.2418503653809263
998     0.5326372574991043
999     3.1891046467482984
1000     1.439804178316969
95% CI for the 10th customer's wait: (1.2783226033010433, '+/-', 0.08617532063280078)
```

Problem 2.

Add an upper bound barrier B to the original Asian Model, if the stock price goes above the Barrier the value of stock is 0, which limit the most money you can get from the option to B – K. An indicator is added to the original model. First, record the X(t) in a list, if the maximum of X(t) in the list is below B, the value of the option is the same as Asian Options, otherwise the value is 0. When computing the value of stock, the indicator is added to the multiplier, int(max(X(t) < B) = 1 if true. =0 if false. Run 40000 replications, the mean and 95% confidence interval are computed with CI_95 function.

B = 60, CI 95% = [0.000, 8.23×10^{-5}], mean = 4.09×10^{-5}

B = 65, CI 95% = [0.018, 0.022], mean = 0.020

B = 70, CI 95% = [0.156,0.170], mean = 0.163

Increasing the barrier will increase the expected value of the option since the barrier limit the benefit one can get from the option.

Code and results shown below.

```
import numpy as np
def CI_95(data):
    a = np.array(data)
    n = len(a)
    m = np.mean(a)
    sd = np.std(a,ddof=1)
    hw = 1.96*sd / np.sqrt(n)
    return m, [m-hw,m+hw]
Maturity = 1.0
InterestRate = 0.05
Sigma = 0.3
InitialValue = 50.0
StrikePrice = 55.0
B = 60 # Set the barrier value B = 65 B = 70
Steps = 64
Interval = Maturity / Steps
Sigma2 = Sigma * Sigma / 2

np.random.seed(1)
Replications = 40000
ValueList = [] # List to keep the option value for each sample path
for i in range(0,Replications):
    Sum = 0.0
    Xt = [] # add a list to store stock price X at each step
    X = InitialValue
    for j in range(0,Steps):
        Z = np.random.standard_normal(1)
        X = X * np.exp((InterestRate - Sigma2) * Interval +
                       Sigma * np.sqrt(Interval) * Z)
```

```
        Sum = Sum + X
        Xt.append(X) # record the stock price at step j to the list Xt
    Value = np.exp(-InterestRate * Maturity) *int(max(Xt) < B) *
max(Sum/Steps - StrikePrice, 0) # add the indicator function
    ValueList.append(float(Value))
print ("Mean and CI:", CI_95(ValueList))
```

B = 60, result shown below, the lower end should be 0, it's a rounding error from python.

```
Mean and CI: (4.092375775163377e-05, [-4.4481329553437785e-07, 8.229232879880192e-05])
```

B = 65, result shown below

```
Mean and CI: (0.020044237428748078, [0.018274856586630908, 0.021813618271187076])
```

B = 70, result shown below

```
Mean and CI: (0.16320852410715334, [0.15648048132926246, 0.16993656688504422])
```

Problem 3.

Change the number of servers to s, change the arrival rate λ to s, so MeanTBA = 1/s. Service time = SimRNG.Erlang(Phases, MeanST, 2) stays the same as M/G/1 model.

Define the arrival function: when a customer arrives, add the customer to the queue. Check the number of busy servers. If the number of busy servers < the total number of servers, take the customer from the queue and ask the server to seize the customer, and remove the customer from the queue. The seize function increase the number of busy servers by 1. Then schedule the service time, and the 'EndOfService' function with SchedulePlus in the class of SimFuntions.

Define the EndOfService function: at the end of service, and record the customers total time in the system. Free 1 server, which decrease the number of busy servers by 1. If there are customers in the queue, seize 1 customer and remove 1 customer in the queue, then schedule the service time and the 'EndOfService' function.

Do 10 replications with the warmup time of 500 and the total run time of 5500. Record the total time in system as Wait, number of customers in queue, and the busy serves and compute the mean.

(a).

Result:

With 10 servers, mean arrival rate = 10:

- Estimated Expected System Time: 0.9142568908659282
- Estimated Expected Average Utilization: 0.8000989287730753

With 20 servers, mean arrival rate = 20

- Estimated Expected System Time: 0.8379191428554629
- Estimated Expected Average utilization: 0.800951322570904

With 30 servers, mean arrival rate = 30

- Estimated Expected System Time: 0.8178473853292024
- Estimated Expected Average utilization: 0.8014010825528823

(b).

As the number of servers and arrival rate increases simultaneously, the utilization of the system does not change much, but the average system time for customers decreased a lot. Considering the mean service time of 0.8, the average time of customer staying in queue is significantly reduced. Therefore, a stable system performs better as the size of system increases.

Code shown and run result shown below.

```
import SimFunctions
import SimRNG
import SimClasses
import numpy as np
import pandas

ZSimRNG = SimRNG.InitializeRNSeed()

Queue = SimClasses.FIFOQueue()
Wait = SimClasses.DTStat()
Server = SimClasses.Resource()
Calendar = SimClasses.EventCalendar()

TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []

TheDTStats.append(Wait)
TheQueues.append(Queue)
TheResources.append(Server)

s = 10 # the number of servers s = 20 s = 30
Server.SetUnits(s)
MeanTBA = 1/s # lamda = s, so arrival interval is 1/s
```

```
MeanST = 0.8
Phases = 3
RunLength = 5500
WarmUp = 500

AllWaitMean = []
AllQueueMean = []
AllQueueNum = []
AllServerMean = []
print("Rep", "Average Wait", "Average Number in Queue", "Number Remaining
in Queue", "Server Utilization")


def Arrival():
    SimFunctions.Schedule(Calendar, "Arrival", SimRNG.Expon(MeanTBA, 1))
    Customer = SimClasses.Entity()
    Queue.Add(Customer)

    if Server.Busy < s:
        Server.Seize(1)
        NextCustomer = Queue.Remove() # customer leaves the queue if
served
        SimFunctions.SchedulePlus(Calendar, "EndOfService",
SimRNG.Erlang(Phases, MeanST, 2), NextCustomer)


def EndOfService(DepartingCustomer): # give an object to the EndOfService
function to record total time in system
    Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime) # Record
the customer's total time in system
    Server.Free(1)

    if Queue.NumQueue() > 0:
        Server.Seize(1)
        NextCustomer = Queue.Remove()
        SimFunctions.SchedulePlus(Calendar, "EndOfService",
SimRNG.Erlang(Phases, MeanST, 2), NextCustomer)

for reps in range(0, 10, 1):

    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats,
TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Arrival", SimRNG.Expon(MeanTBA, 1))
    SimFunctions.Schedule(Calendar, "EndSimulation", RunLength)
    SimFunctions.Schedule(Calendar, "ClearIt", WarmUp)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Arrival":
        Arrival()
    elif NextEvent.EventType == "EndOfService":
        EndOfService(NextEvent.WhichObject)
```

```python
    elif NextEvent.EventType == "ClearIt":
        SimFunctions.ClearStats(TheCTStats, TheDTStats)

    while NextEvent.EventType != "EndSimulation":
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Arrival":
            Arrival()
        elif NextEvent.EventType == "EndOfService":
            EndOfService(NextEvent.WhichObject)
        elif NextEvent.EventType == "ClearIt":
            SimFunctions.ClearStats(TheCTStats, TheDTStats)

    AllWaitMean.append(Wait.Mean())
    AllQueueMean.append(Queue.Mean())
    AllQueueNum.append(Queue.NumQueue())
    AllServerMean.append(Server.Mean() / s) # compute the utilization of
servers
    print(reps + 1, Wait.Mean(), Queue.Mean(), Queue.NumQueue(),
Server.Mean() / s)

# output results

print("Estimated Expected System Time:", np.mean(AllWaitMean))
print("Estimated Expected Average queue-length:", np.mean(AllQueueMean))
print("Estimated Expected Average utilization:", np.mean(AllServerMean))
```

Result for s = 10

```
Rep Average Wait Average Number in Queue Number Remaining in Queue Server Utilization
1 0.9202356725313969 1.2000960658043573 0 0.8065911182099589
2 0.9268931060083561 1.2693312845365294 2 0.8072707830487772
3 0.9160865623123235 1.1587811511026158 0 0.7993796573767612
4 0.9049704439640434 1.0960778785625718 0 0.7947067288563858
5 0.9121300977355 1.1090569030381279 0 0.7985670782397785
6 0.9119192992556105 1.103541131808623 0 0.8019876742023279
7 0.9136588806866439 1.1230596538814492 0 0.7986958343570124
8 0.9131362669154276 1.1582427509552682 0 0.7954551836329407
9 0.9118774557079792 1.1314086020527296 0 0.7991660217115621
10 0.9116611235419994 1.0905266081232479 0 0.7991692080952495
Estimated Expected System Time: 0.9142568908659282
Estimated Expected Average queue-length: 1.144012202986552
Estimated Expected Average utilization: 0.8000989287730753
```

Result for s = 20

```
Rep Average Wait Average Number in Queue Number Remaining in Queue Server Utilization
1 0.8417835163162608 0.8042935641461118 0 0.8078424789977008
2 0.833484581443263 0.7200838122719014 0 0.796400518728258
3 0.8364970531602192 0.6999915780127448 0 0.799915092165998
4 0.836830011830332 0.7471878464009779 0 0.7978438924812827
5 0.8369507709094374 0.7312634927799034 0 0.7992875437905234
6 0.8375569452857682 0.73968505184186 0 0.8029459300397915
7 0.8412376214950555 0.7990181729301109 0 0.8075246399489562
8 0.8394329183800244 0.7914370260627397 0 0.800831506254821
9 0.8338987154283939 0.6913327998769849 4 0.7947631371508226
10 0.8415192943058735 0.7774663757021869 1 0.8021584861508835
Estimated Expected System Time: 0.8379191428554629
Estimated Expected Average queue-length: 0.7501759720025521
Estimated Expected Average utilization: 0.800951322570904
```

Result for s = 30

```
Rep Average Wait Average Number in Queue Number Remaining in Queue Server Utilization
1 0.8197436819514927 0.5499791681687742 0 0.804904274662523
2 0.8155908463342633 0.47942178468917646 0 0.7978080725602138
3 0.816378871950401 0.522543885964659 0 0.7987993394924856
4 0.819122708443033 0.530695870384765 1 0.8023193340460539
5 0.8199944602803447 0.5731330804349724 0 0.8060983009456096
6 0.8155533323219593 0.4772743392096081 0 0.7954815979625814
7 0.818834430985118 0.5224667030128987 7 0.8018697172188999
8 0.8189586977935446 0.5388851585578943 0 0.8029615849516454
9 0.8160727923841516 0.49821241990258125 0 0.8010229427671293
10 0.8182240308477168 0.5035831838211123 0 0.8027456609216815
Estimated Expected System Time: 0.8178473853292024
Estimated Expected Average queue-length: 0.5196195594146442
Estimated Expected Average utilization: 0.8014010825528823
```

Problem 4

The approach to set the customer return with p = 0.1 is to generate a random number U(0,1) with SimRNG.Uniform(0, 1, 1)at the 'EndOfService' event, if the random number < 0.1 then schedule the return for the customer with rate of Expon(2). The return of customer is done by defining a Return function. Most part of the Return function is the same as the Arrival function except that the schedule is named as 'Return', and the return rate is Expon(2).

Simulation conclusion:

Estimate for the expected steady-state number of customers in system is 5.924664041541135

Code and run result show below.

```
import SimFunctions
import SimRNG
import SimClasses
import numpy as np
import pandas


ZSimRNG = SimRNG.InitializeRNSeed()

Queue = SimClasses.FIFOQueue()
Wait = SimClasses.DTStat()
Server = SimClasses.Resource()
Calendar = SimClasses.EventCalendar()

TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []

TheDTStats.append(Wait)
TheQueues.append(Queue)
TheResources.append(Server)

Server.SetUnits (1)
MeanTBA = 1
MeanST = 0.8
Phases = 3
RunLength = 55000
WarmUp = 5000

AllWaitMean = []
AllQueueMean = []
AllQueueNum = []
AllServerMean = []
print ("Rep", "Average Wait", "Average Number in Queue", "Number Remaining
in Queue", "Server Utilization")
```

```
def Arrival():
    SimFunctions.Schedule(Calendar,"Arrival",SimRNG.Expon(MeanTBA, 1))
    Customer = SimClasses.Entity()
    Queue.Add(Customer)

    if Server.Busy == 0:
        Server.Seize(1)

SimFunctions.Schedule(Calendar,"EndOfService",SimRNG.Erlang(Phases,MeanST,
2))


def Return():
    SimFunctions.Schedule(Calendar, "Return", SimRNG.Expon(2, 1)) # the
customer return schedule
    ReturnCustomer = SimClasses.Entity()
    Queue.Add(ReturnCustomer)

    if Server.Busy == 0:
        Server.Seize(1)
        SimFunctions.Schedule(Calendar, "EndOfService",
SimRNG.Erlang(Phases, MeanST, 2))

def EndOfService():
    DepartingCustomer = Queue.Remove()
    Wait.Record(SimClasses.Clock - DepartingCustomer.CreateTime)
    if Queue.NumQueue() > 0:

SimFunctions.Schedule(Calendar,"EndOfService",SimRNG.Erlang(Phases,MeanST,
2))
    else:
        Server.Free(1)

    if SimRNG.Uniform(0, 1, 1) < 0.1: # after the service, p = 0.1 that
customer will return
        Return()

for reps in range(0,10,1):


SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,TheDTStats,The
Resources)
    SimFunctions.Schedule(Calendar,"Arrival",SimRNG.Expon(MeanTBA, 1))
    SimFunctions.Schedule(Calendar,"EndSimulation",RunLength)
    SimFunctions.Schedule(Calendar,"ClearIt",WarmUp)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Arrival":
        Arrival()
    elif NextEvent.EventType == "EndOfService":
        EndOfService()
```

```python
    elif NextEvent.EventType == "ClearIt":
        SimFunctions.ClearStats(TheCTStats,TheDTStats)

    while NextEvent.EventType != "EndSimulation":
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Arrival":
            Arrival()
        elif NextEvent.EventType == "EndOfService":
            EndOfService()
        elif NextEvent.EventType == "ClearIt":
            SimFunctions.ClearStats(TheCTStats,TheDTStats)


    AllWaitMean.append(Wait.Mean())
    AllQueueMean.append(Queue.Mean())
    AllQueueNum.append(Queue.NumQueue())
    AllServerMean.append(Server.Mean())
    print (reps+1, Wait.Mean(), Queue.Mean(), Queue.NumQueue(),
Server.Mean())

# output results

print("Estimated Expected Average wait:",np.mean(AllWaitMean))
print("Estimated Expected Average queue-length:", np.mean(AllQueueMean))
print("Estimated Expected Average utilization:",np.mean(AllServerMean))
```

Run Result:

```
Rep Average Wait Average Number in Queue Number Remaining in Queue Server Utilization
1 6.132521332765881 6.876473576153403 0 0.899042161581794
2 5.656896223709687 6.270732132663308 12 0.8880047787288727
3 5.072335786562361 5.6204760341618805 3 0.8865413421952608
4 5.152721589048437 5.716401851730538 4 0.8815589218281895
5 5.556416630739592 6.156897047173882 5 0.8915190547102892
6 5.211319668702209 5.788956397425572 1 0.8901189966142395
7 5.169885685309436 5.723237504526151 2 0.8839275831058576
8 5.112591090921315 5.647743008996316 6 0.8840451829003511
9 4.91210026598368 5.424800071840622 4 0.8853645730175664
10 5.423032098803933 6.020922790739679 10 0.8919750669555233
Estimated Expected Average wait: 5.339982037254653
Estimated Expected Average queue-length: 5.924664041541135
Estimated Expected Average utilization: 0.8882097661637944
```

Problem 5.

The simulation of the model is very similar to M/G/1 queue simulation. ‘Arrival’ is replaced with ‘Callin’, ‘EndOfService’ is replaced with ‘Endcall’. A warmup is not necessary for the model to determine the queue capacity and utilization, so it’s removed from the model. Upon ‘Callin’ event, use SimRNG.Uniform(0, 1, 1) < 0.1 to represent p = 0.1 that callers leaves the queue if not served immediately. Create a list called QueueLength to record the number of callers in queue after each ‘Callin’ event. The Wait uses classes DTStat to record the wait time in queue of each caller. The Lost (used in the 2nd and 3rd run) uses classes DTStat to record the calls lost if the caller calls in and the queue capacity is reached, note that the caller leaves the queue as well.

We will use min as run unit. MeanTBA: 10; Arrival: Expon(10) . Service: uniform (5,12). Server: 1 operator. Queue: p1 = 0.9: FIFOQueue, p2 = 0.1 leave system. Stream 1 of SimRNG is used for all random numbers. Run 10 replications of the model with run length of 500000 min.

Simulation conclusion: A queue capacity of 6 is required to ensure less than 5% loss of calls. 3.2% of calls are lost with a queue capacity of 6, the utilization of operator is 77.7% with a queue capacity of 6.

1st run of model is to determine the queue capacity: (The calls lost parts are commented). Run the model with no queue capacity, so no callers are lost. Then record the number of callers in queue at each arrival in a list QueueLength. The 95% quantile of the sorted QueueLength gives the required capacity of less than 5% calls lost.

```python
import SimClasses
import SimFunctions
import SimRNG
import math
import pandas
import numpy as np

ZSimRNG = SimRNG.InitializeRNSeed()

Queue = SimClasses.FIFOQueue()
Wait = SimClasses.DTStat()
Lost = SimClasses.DTStat()
Server = SimClasses.Resource()
Calendar = SimClasses.EventCalendar()

TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []

TheDTStats.append(Wait)
TheDTStats.append(Lost)
```

```
TheQueues.append(Queue)
TheResources.append(Server)


Server.SetUnits (1)
MeanTBA = 10
# ST = Unifrom (5,12)
Phases = 3
RunLength = 500000 # unit in min

QueueLength = [] # QueueLength records the number of queue

AllWaitMean = []
AllQueueMean = []
AllQueueNum = []
AllServerMean = []
AllLostMean = []
Capacity = [] # 95% of quantile of sorted QueueLength

print ("Rep", "Average Wait", "Server Utilization", "95% Quantile of
Queue", "Total # of Callers")


def Callin():
    global QueueLength
    SimFunctions.Schedule(Calendar, "Callin", SimRNG.Expon(MeanTBA, 1))
    Caller = SimClasses.Entity()
    Queue.Add(Caller)

    if Server.Busy == 0:
        Server.Seize(1)
        NextCaller = Queue.Remove()
        Wait.Record(SimClasses.Clock - NextCaller.CreateTime)
        SimFunctions.SchedulePlus(Calendar, "EndCall", SimRNG.Uniform(5,
12, 1),NextCaller)
 #       Lost.Record(0) # Lost count 0, caller not lost
    elif SimRNG.Uniform(0, 1, 1) < 0.1: # 10% chance the caller hangup if
not served immediately
        Queue.Remove()
 #       Lost.Record(0) # Lost count 0, caller not lost
 #    elif Queue.NumQueue() > 5:
 #        Queue.Remove()
 #        Lost.Record(1) # Lost count 1, calls lost
    QueueLength.append(Queue.NumQueue()) # Record the Queue length after
each Callin

def EndCall():
    Server.Free(1)
    if Queue.NumQueue() > 0:
        Server.Seize(1)
        NextCaller = Queue.Remove()
        Wait.Record(SimClasses.Clock - NextCaller.CreateTime)
```

```
        SimFunctions.SchedulePlus(Calendar, "EndCall", SimRNG.Uniform(5,
12, 1),NextCaller)


for reps in range(0, 10, 1):

    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats,
TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Callin", SimRNG.Expon(MeanTBA, 1))
    SimFunctions.Schedule(Calendar, "EndSimulation", RunLength)

    QueueLength = []

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Callin":
        Callin()
    elif NextEvent.EventType == "EndCall":
        EndCall()

    while NextEvent.EventType != "EndSimulation":
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Callin":
            Callin()
        elif NextEvent.EventType == "EndCall":
            EndCall()

    AllWaitMean.append(Wait.Mean())
    AllServerMean.append(Server.Mean())
    #AllLostMean.append(Lost.Mean())
    Quantile95 = int(np.ceil(0.95*len(QueueLength)))
    Capacity.append(np.sort(QueueLength)[Quantile95])
    print(reps + 1,"   ", Wait.Mean(), "   ",Server.Mean(), "   ",Capacity[-
1],"   ",len(QueueLength)), #Lost.Mean())

# output results
print("Estimated required queue capacity for less than 5% loss of callers
is:",np.mean(Capacity))
#print("Estimated Expected Average utilization:", np.mean(AllLostMean))
print("Estimated Expected Average wait:", np.mean(AllWaitMean))
print("Estimated Expected Average utilization:", np.mean(AllServerMean))
```

Result Shows that queue capacity should be greater than 6 to have less than 5% loss of callers.

```
Rep Average Wait Server Utilization 95% Quantile of Queue Total # of Callers
1    13.632690763228833    0.7852371441511322    6    50338
2    13.97800852305605    0.7872215875006471    6    50099
3    13.163673843137113    0.7814426105068447    6    49795
4    13.894261290153041    0.7864663179236516    6    50065
5    13.339534490333836    0.7812208904943466    6    50060
6    13.62244499918022    0.7875771425793476    6    50269
7    14.147831086204006    0.7859365428106962    6    50180
8    13.247655826716544    0.7806301435129086    6    49771
9    13.876713622677338    0.7857590876677454    6    50089
10    14.119173236261553    0.783653606755232    6    50071
Estimated required queue capacity for less than 5% loss of callers is: 6.0
Estimated Expected Average wait: 13.702198768094854
Estimated Expected Average utilization: 0.7845145073902552
```

2nd run is to check the calls lost with the queue capacity of 6, Uncomment the callers loss parts with capacity of 6, and run the model to check calls lost. Use class DTStat to estimate the fraction of calls lost (record a 0 for calls not lost and record a 1 for those that are lost).

```
import SimClasses
import SimFunctions
import SimRNG
import math
import pandas
import numpy as np

ZSimRNG = SimRNG.InitializeRNSeed()

Queue = SimClasses.FIFOQueue()
Wait = SimClasses.DTStat()
Lost = SimClasses.DTStat()
Server = SimClasses.Resource()
Calendar = SimClasses.EventCalendar()

TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []

TheDTStats.append(Wait)
TheDTStats.append(Lost)
TheQueues.append(Queue)
TheResources.append(Server)


Server.SetUnits (1)
MeanTBA = 10
# ST = Unifrom (5,12)
```

```python
Phases = 3
RunLength = 500000 # unit in min

QueueLength = [] # QueueLength records the number of queue

AllWaitMean = []
AllQueueMean = []
AllQueueNum = []
AllServerMean = []
AllLostMean = []
Capacity = [] # 95% of quantile of sorted QueueLength

print ("Rep", "Average Wait", "Server Utilization", "95% Quantile of Queue", "Total # of Callers" , "Callers Loss")


def Callin():
    global QueueLength
    SimFunctions.Schedule(Calendar, "Callin", SimRNG.Expon(MeanTBA, 1))
    Caller = SimClasses.Entity()
    Queue.Add(Caller)

    if Server.Busy == 0:
        Server.Seize(1)
        NextCaller = Queue.Remove()
        Wait.Record(SimClasses.Clock - NextCaller.CreateTime)
        SimFunctions.SchedulePlus(Calendar, "EndCall", SimRNG.Uniform(5, 12, 1),NextCaller)
        Lost.Record(0) # Lost count 0, caller not lost
    elif SimRNG.Uniform(0, 1, 1) < 0.1: # 10% chance the caller hangup if not served immediately
        Queue.Remove()
        Lost.Record(0) # Lost count 0, caller not lost
    elif Queue.NumQueue() > 6: # Choose the capacity of 6
        Queue.Remove()
        Lost.Record(1) # Lost count 1, calls lost
    QueueLength.append(Queue.NumQueue()) # Record the Queue length after each Callin

def EndCall():
    Server.Free(1)
    if Queue.NumQueue() > 0:
        Server.Seize(1)
        NextCaller = Queue.Remove()
        Wait.Record(SimClasses.Clock - NextCaller.CreateTime)
        SimFunctions.SchedulePlus(Calendar, "EndCall", SimRNG.Uniform(5, 12, 1),NextCaller)


for reps in range(0, 10, 1):

    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats,
```

```
TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Callin", SimRNG.Expon(MeanTBA, 1))
    SimFunctions.Schedule(Calendar, "EndSimulation", RunLength)

    QueueLength = []

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Callin":
        Callin()
    elif NextEvent.EventType == "EndCall":
        EndCall()

    while NextEvent.EventType != "EndSimulation":
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Callin":
            Callin()
        elif NextEvent.EventType == "EndCall":
            EndCall()

    AllWaitMean.append(Wait.Mean())
    AllServerMean.append(Server.Mean())
    AllLostMean.append(Lost.Mean())
    Quantile95 = int(np.ceil(0.95*len(QueueLength)))
    Capacity.append(np.sort(QueueLength)[Quantile95])
    print(reps + 1,"  ", Wait.Mean(), "  ",Server.Mean(), "  ",Capacity[-
1],"  ",len(QueueLength), Lost.Mean())

# output results
print("Estimated required queue capacity for less than 5% loss of callers
is:",np.mean(Capacity))
print("Estimated Expected Average loss of callers:", np.mean(AllLostMean))
print("Estimated Expected Average wait:", np.mean(AllWaitMean))
print("Estimated Expected Average utilization:", np.mean(AllServerMean))
```

Result shows that only 3.2% calls are lost with a queue capacity of 6, and the suggested capacity changes to 5 with a queue capacity of 6, so let’s try a queue capacity of 5 and run the model again.

```
Rep Average Wait Server Utilization 95% Quantile of Queue Total # of Callers Callers loss
1    11.296577036723356    0.7786487388645846    5    50379 0.03206166202474253
2    11.230756777042554    0.7794099247966633    5    50194 0.03433087187459515
3    10.935677187816792    0.7730830386939759    5    49757 0.02974901261307173
4    10.989826734603982    0.778047918244609    5    50004 0.03352891869237217
5    10.932649904356365    0.7762967023085777    5    50163 0.02914169111082770 2
6    11.332372791664218    0.780360541443879    5    50309 0.03324009627268588
7    11.437026218834017    0.7786329200185022    5    50272 0.03525476067936181
8    10.746343241928729    0.772374969065939    5    49703 0.02709784623250463 2
9    11.17239831320318    0.7779853666590723    5    50031 0.03272445620602853
10    11.100829520053862    0.7772790398969784    5    50151 0.031959359526718537
Estimated required queue capacity for less than 5% loss of callers is: 5.0
Estimated Expected Average loss of callers: 0.03190886752329087
Estimated Expected Average wait: 11.117445772622705
Estimated Expected Average utilization: 0.7772119159992781
```

Run the model with queue capacity of 5.

```python
import SimClasses
import SimFunctions
import SimRNG
import math
import pandas
import numpy as np

ZSimRNG = SimRNG.InitializeRNSeed()

Queue = SimClasses.FIFOQueue()
Wait = SimClasses.DTStat()
Lost = SimClasses.DTStat()
Server = SimClasses.Resource()
Calendar = SimClasses.EventCalendar()

TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []

TheDTStats.append(Wait)
TheDTStats.append(Lost)
TheQueues.append(Queue)
TheResources.append(Server)


Server.SetUnits (1)
MeanTBA = 10
# ST = Unifrom (5,12)
Phases = 3
RunLength = 500000 # unit in min
```

```python
QueueLength = [] # QueueLength records the number of queue

AllWaitMean = []
AllQueueMean = []
AllQueueNum = []
AllServerMean = []
AllLostMean = []
Capacity = [] # 95% of quantile of sorted QueueLength

print ("Rep", "Average Wait", "Server Utilization", "95% Quantile of
Queue", "Total # of Callers", "Callers Loss")


def Callin():
    global QueueLength
    SimFunctions.Schedule(Calendar, "Callin", SimRNG.Expon(MeanTBA, 1))
    Caller = SimClasses.Entity()
    Queue.Add(Caller)

    if Server.Busy == 0:
        Server.Seize(1)
        NextCaller = Queue.Remove()
        Wait.Record(SimClasses.Clock - NextCaller.CreateTime)
        SimFunctions.SchedulePlus(Calendar, "EndCall", SimRNG.Uniform(5,
12, 1),NextCaller)
        Lost.Record(0) # Lost count 0, caller not lost
    elif SimRNG.Uniform(0, 1, 1) < 0.1: # 10% chance the caller hangup if
not served immediately
        Queue.Remove()
        Lost.Record(0) # Lost count 0, caller not lost
    elif Queue.NumQueue() > 5: # Choose the capacity of 5
        Queue.Remove()
        Lost.Record(1) # Lost count 1, calls lost
    QueueLength.append(Queue.NumQueue()) # Record the Queue length after
each Callin

def EndCall():
    Server.Free(1)
    if Queue.NumQueue() > 0:
        Server.Seize(1)
        NextCaller = Queue.Remove()
        Wait.Record(SimClasses.Clock - NextCaller.CreateTime)
        SimFunctions.SchedulePlus(Calendar, "EndCall", SimRNG.Uniform(5,
12, 1),NextCaller)


for reps in range(0, 10, 1):

    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats,
TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Callin", SimRNG.Expon(MeanTBA, 1))
    SimFunctions.Schedule(Calendar, "EndSimulation", RunLength)
```

```python
    QueueLength = []

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    if NextEvent.EventType == "Callin":
        Callin()
    elif NextEvent.EventType == "EndCall":
        EndCall()

    while NextEvent.EventType != "EndSimulation":
        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Callin":
            Callin()
        elif NextEvent.EventType == "EndCall":
            EndCall()

    AllWaitMean.append(Wait.Mean())
    AllServerMean.append(Server.Mean())
    AllLostMean.append(Lost.Mean())
    Quantile95 = int(np.ceil(0.95*len(QueueLength)))
    Capacity.append(np.sort(QueueLength)[Quantile95])
    print(reps + 1,"   ", Wait.Mean(), "   ",Server.Mean(), "   ",Capacity[-
1],"   ",len(QueueLength), Lost.Mean())

# output results
print("Estimated required queue capacity for less than 5% loss of callers
is:",np.mean(Capacity))
print("Estimated Expected Average loss of callers:", np.mean(AllLostMean))
print("Estimated Expected Average wait:", np.mean(AllWaitMean))
print("Estimated Expected Average utilization:", np.mean(AllServerMean))
```

Result shows that calls lost is 5.1% > 5%, so the queue capacity need to be 6 to ensure less than 5% calls lost.

```
Rep Average Wait Server Utilization 95% Quantile of Queue Total # of Callers Callers Loss
1    10.196499723575267    0.774188307584395    5    50431 0.05034581593868777
2    10.207916747553837    0.7758937681952469    5    50272 0.053631703962995374
3    10.057122425583934    0.7706918242141091    5    49899 0.04953675108091415
4    9.983375086036219    0.7714034199563534    5    49992 0.05440207011274721
5    9.85550005030239    0.7671403103277359    5    49944 0.04962764009278477
6    10.279514410100916    0.7762081246414184    5    50390 0.053800437910541135
7    10.230119157053084    0.771939694870472    5    50182 0.05674418604651163
8    9.78803162591401    0.7659582771684502    5    49579 0.04583642204101804
9    10.160769202115109    0.7697819277584076    5    49843 0.04974196356401169
10    9.959459376416993    0.7716690702831231    5    50188 0.052040503221847195
Estimated required queue capacity for less than 5% loss of callers is: 5.0
Estimated Expected Average loss of callers: 0.05157074939720589
Estimated Expected Average wait: 10.071830780465174
Estimated Expected Average utilization: 0.7714874724999712
```

Problem 6

$$\gamma^2 = \sigma^2\left(1+2\sum_{k=1}^{\infty}\rho_k\right)$$

For AR(1) model

$$\boxed{\sigma^2 = \frac{\sigma^2}{1-\varphi^2}}$$

$$\boxed{\rho_k = \varphi^k}$$

$$\Rightarrow \frac{\sigma^2}{1-\varphi^2}\left(1+2\sum_{k=1}^{\infty}\varphi^k\right)$$

$$0<\varphi<1 \Rightarrow \frac{\sigma^2}{1-\varphi^2}\left(1+2\frac{\varphi}{1-\varphi}\right)$$

$$\Rightarrow \gamma^2 = \frac{\sigma^2}{1-\varphi^2}\cdot\frac{1+\varphi}{1-\varphi}$$

$$\Rightarrow \gamma^2 = \frac{\sigma^2}{|1-\varphi|^2}$$