Homework #4 MIE 1613

David Islip

MIE Flextime

In [72]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
import math
import SimFunctions
import SimRNG
import SimClasses
sns.set_style('whitegrid')
%matplotlib inline
```

Problem #1 : Steady State Waiting time Estimation for M/G/1 Queue

The problem is a simulation of the M/G/1 queue using Lindley's equation:

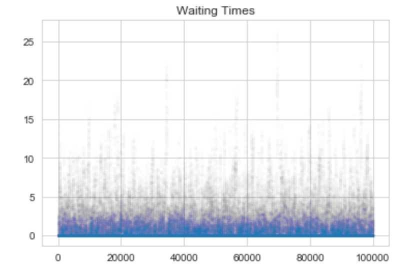
$$Y_i = \max\{0, Y_{i-1} + X_{i-1} - A_i\}, i = 1, 2, \ldots$$

where Y_i is the ith customer's waiting time, X_i is that customer's service time, and A_i is the interarrival time between customers $i-1$ and i

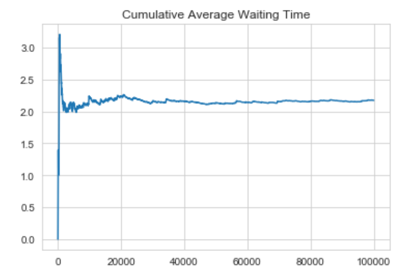
Step 1: Determine the warm up

In [2]:
```
#Code Adapted from the python notebook from class

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

m = 100000 #length
d = 0 #warmup of zero for now

MeanTBA = 1.0 # average interarrival time
MeanST = 0.8 # average service time

def simulate_mg1(m,d, MeanTBA, MeanST):
    np.random.seed(1)
    Y = np.zeros(m-d)
    for i in range(1,d+1,1):
        print(i)
        A = np.random.exponential(MeanTBA, 1)
        X = np.sum(np.random.exponential(MeanST/3,3))
        Y[i] = max(0, Y[i-1] + X - A)

    for i in range(1+d,m,1):
        A = np.random.exponential(MeanTBA, 1)
        X = np.sum(np.random.exponential(MeanST/3,3))
        Y[i] = max(0, Y[i-1] + X - A)
    return Y
Y = simulate_mg1(m,d, MeanTBA, MeanST)
```


In [3]:
```
plt.title("Waiting Times")
plt.plot(Y,'.' ,alpha = 0.005);
```

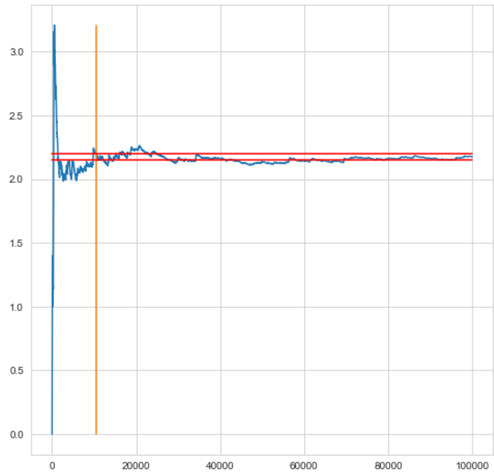


In [4]:
```
#cumulative Average
plt.title("Cumulative Average Waiting Time")
Y_j = Y.cumsum()/np.arange(1,len(Y)+1)
plt.plot(Y_j);
```



In [5]:
```
# detecting when the average does not not deviate from the final average by more than
#the specifed tolerance
tol = 0.01*Y_j[-1]
i=0
m = 100
j = 0
while i < len(Y):
    if abs(Y_j[i] - Y_j[-1]) < tol:
        j = j + 1
        if j >= m:
            break
    else:
        j = 0
    i = i + 1
```

In [6]:
```
plt.figure(figsize = (8,8))
plt.plot(Y_j);
plt.plot([i,i],[0,np.max(Y_j)]);
plt.plot([0,len(Y_j)],[Y_j[-1]+tol,Y_j[-1]+tol], 'red');
plt.plot([0,len(Y_j)],[Y_j[-1]-tol,Y_j[-1]-tol], 'red');
```



Drop the first observations up to index d.

From the textbook:

1. Divide the remaining m−d observations into from 10 ≤ k ≤ 30 batches, looking for a value of k that divides m − d close to evenly (if there are data left over, delete from the beginning).
2. Compute the sample mean and form the batch means confidence interval (8.17)

In [7]:
```
d = i
Y_warm = Y[d:]
#Choose 20 batches for approx 5000 obs per batch estimate
k = 20
batches = np.array_split(Y_warm, k)
q= 0.8
quantiles = [np.quantile(batches[i], q) for i in range(len(batches))]
```


In [8]:
```
quantiles
```

Out[8]:
```
[3.797904887289595,
 3.809169173830547,
 3.7913387039923063,
 3.1534725475813654,
 3.833335753413503,
 3.570918537466137,
 3.6439724762647483,
 3.2691966493436073,
 3.967381245234501,
 3.627543555651312,
 4.005163588715895,
 3.566109267260474,
 3.556233593691834,
 4.265966082084617,
 3.5900237641242576,
 3.6646907480739483,
 4.565980422551518,
 3.0872559953553544,
 3.3636521748132924,
 4.387382311072847]
```

In [9]:
```
t_mean_confidence_interval(quantiles,0.05)
```

Out[9]:
```
(3.7258345738905825, '+/-', 0.17939074631134 32)
```

Problem #2

Part a: Simulation Logic

The code below provides an implementation of a discrete event based simulation model for the inventory system considered in the question. The code is broken down into two components.

Part 1: Problem Set Up

- Define the statistics of interest and set them up in using the boilerplate code from previous experiments.
- Define the distribution function for the random demand amount
- Specify problem parameters such as costs and probability of times

Part 2: Simulation Function

- Specify the event functions and helper functions
- Replication loop
- For the case of one replication output the sample path information (for sense checking)

Event Functions

Three events are outlined: i) The arrival of new demand, ii) Placing Orders, iii) Arriving Orders. Please read the doc strings in the sub functions inside the function:

```
simulate_inventory
```

for more information.

In [138]:
```
s = 20 #policy parameter
S = 40 #policy parameter
N = 1 #number of replications
T = 120 #length of simulation in months
```

In [151]:

```python
#Part 1 Problem Set up
##STATISTICS OF INTEREST (USED TO CALCULATE THE EXPECTED COST)
I = SimClasses.CTStat()
I_neg = SimClasses.CTStat()
I_pos = SimClasses.CTStat()
Order_Cost = SimClasses.DTStat()
Calendar = SimClasses.EventCalendar()

##Setting up the statistics
TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []
TheCTStats.append(I)
TheCTStats.append(I_neg)
TheCTStats.append(I_pos)
TheDTStats.append(Order_Cost)


#Part 2 Function Definition
def simulate_inventory(s,S,N,T):

    ##SPECIFYING THE PROBABILITY DISTRIBUTION FOR
    ##THE DEMAND FUNCTION
    Demand_Elements = [1,2,3,4]
    Demand_Probabilities = np.array([1/6, 1/3, 1/3, 1/6])
    distrib = list(Demand_Probabilities.cumsum())

    ##MODEL PARAMETERS GIVEN IN THE QUESTION
    start = 60 #starting inventory
    K = 32 #setup cost
    i = 3 #incremental cost
    MeanTBD = 0.1 #time between demands
    lowerOLT = 0.5 #order lag time
    upperOLT = 1 #order lag time
    TBO = 1 #time between orders
    pi = 5 #shortage cost
    h = 1 #excess inventory cost

    ## EVENT FUNCTIONS
    def DemandArrival():
        '''This function updates the Inventroy counts
           according to the random discrete distribution
           and schedules the next demand arroval according
           to an exponential distribution'''
        #schedule next demand
        SimFunctions.Schedule(Calendar,"DemandArrival",SimRNG.Expon(MeanTBD, 1))
        #the demand is random iid
        D = SimRNG.Random_integer(distrib,2)
        New_I = I.Xlast - D
        update_inventories(I, I_neg, I_pos, New_I, record)

    def OrderArrival(Z):
        '''Upon an orders arrival the inventory will increase
        this function takes in an amount Z of inventory
        updates the inventory amount'''
        #Update inventory
        New_I = I.Xlast + Z
        update_inventories(I, I_neg, I_pos, New_I, record)

    def ScheduleOrder(s,S):
        '''This function:
        i)  Schedules the next order TBO time units from clock
        ii) Take in the policy parameters s and S to determine the order size.
            if there is an order then the order's arrival is scheduled
        iii) the function returns the order amount'''
        SimFunctions.Schedule(Calendar,"ScheduleOrder",TBO)
        #determine order size
        if I.Xlast < s:
            #schedule the order
            #track the cost
```



```python
            Z = S - I.Xlast
            Order_Cost.Record(K + i*Z)
            SimFunctions.Schedule(Calendar,"OrderArrival",SimRNG.Uniform(lowerOLT,upperOLT,3))
        else:
            Z = 0
        return Z


    #IF THERE IS ONE REPLICATION THE CODE
    #WILL OUTPUT THE INVENTORY SAMPLE PATH
    record = N == 1
    I_hist = []
    I_neg_hist = []
    I_pos_hist = []
    T_hist = []

    def update_inventories(I, I_neg, I_pos, x, record = False):
        #a short cut to update the statistics
        I.Record(x)
        I_pos.Record(np.max([I.Xlast, 0]))
        I_neg.Record(np.max([-1*I.Xlast, 0]))
        #if the record flag is true then the
        #append the statistic values to the historical
        #lists
        if record:
            I_neg_hist.append(I_neg.Xlast)
            I_pos_hist.append(I_pos.Xlast)
            I_hist.append(I.Xlast)
            T_hist.append(SimClasses.Clock)

    ##STORING REPLICATION COST OUTPUTS
    Cost = []

    ##REPLICATION LOOP
    for reps in range(0,N,1):
        SimFunctions.SimFunctionsInit(Calendar,TheQueues,TheCTStats,TheDTStats,TheResources)
        update_inventories(I, I_neg, I_pos, 60, record)
        SimFunctions.Schedule(Calendar,"DemandArrival",SimRNG.Expon(MeanTBD, 1))
        SimFunctions.Schedule(Calendar,"ScheduleOrder",TBO)
        SimFunctions.Schedule(Calendar,"EndSimulation",T)

        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "DemandArrival":
            DemandArrival()
        elif NextEvent.EventType == "ScheduleOrder":
            ScheduleOrder(s,S)
        elif NextEvent.EventType == "EndSimulation":
            EndOfService()

        while NextEvent.EventType != "EndSimulation":
            NextEvent = Calendar.Remove()
            SimClasses.Clock = NextEvent.EventTime
            if NextEvent.EventType == "DemandArrival":
                DemandArrival()
            elif NextEvent.EventType == "ScheduleOrder":
                Z = ScheduleOrder(s,S)
            elif NextEvent.EventType == "OrderArrival":
                OrderArrival(Z)
        #Calculate the cost experienced during that replication
        Total_Cost = pi*I_neg.Mean() + h*I_pos.Mean() + Order_Cost.Sum
        Cost.append(Total_Cost)
    #plot a sample path
    if record:
        plt.figure()
        plt.title("Inventory Sample Path")
        plt.step(T_hist, I_hist);
        plt.figure()
        plt.title("Positive and Negative Components")
        plt.step(T_hist,I_pos_hist, label = '$I^+$');
        plt.step(T_hist,-1*np.array(I_neg_hist), 'orange', label = '$I^-$');
        plt.legend()
```

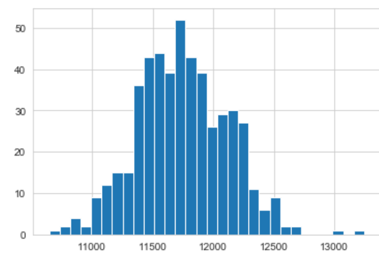
```
    return np.array(Cost)

# Sample Path
simulate_inventory(20,40,N,T)
```

Out[151]:
```
array([12076.1172379 , 11579.06937065, 12014.75442101, 12334.31855657,
       11092.81275541, 11678.08648925, 11409.66368989, 11394.47530357,
       12048.25517417, 12227.1819239 , 11676.40447402, 12037.4107458 ,
       11669.06782494, 11153.62975018, 12153.17222227, 11372.57816615,
       11513.30071949, 12402.21988052, 11369.01710141, 11664.05851795,
       11737.79831876, 12094.46139202, 12704.17751957, 11858.09944783,
       11771.88606619, 11936.7359108 , 11527.62589514, 11886.45579971,
       12018.18707808, 12425.73400573, 11608.19951055, 11670.96279039,
       11731.096491  , 12276.5859279 , 11776.25022994, 11718.64670963,
       12113.52265456, 11079.26874654, 12048.74879641, 11908.88310948,
       11904.37795911, 11699.3835998 , 11602.44549268, 11685.95482346,
       11886.1826057 , 12034.08822576, 11933.34712996, 11613.2154371 ,
       12181.29542285, 11956.95672875])
```

Part b) Confidence Interval (95%) for the expected cost of a policy with $(s;S) = (20;40)$ using 500 replications.

In [152]:
```
N, s, S = 500, 20, 40
costs = simulate_inventory(s,S,N,T)
```

In [153]:
```
plt.hist(costs, bins = 30);
```



In [154]:
```
t_mean_confidence_interval(costs,0.05)
```

Out[154]:
```
(11745.16062866661, '+/-', 33.12155292889283)
```

Therefore the 95% confidence interval is 11740 +- 30

Part c) Subset Selection Method

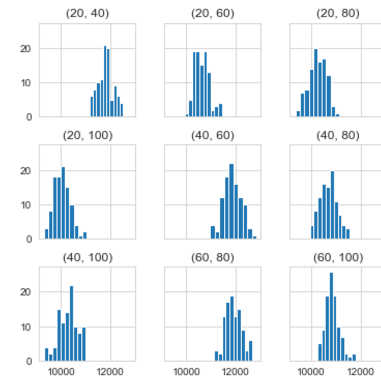
The goal is to deliver a set of feasible scenarios (I) such that the probability that the best scenario (x_b) is in the set is greater than some specified tolerance $1-\alpha$.

i.e deliever I s.t

$$Pr\{x_b \in I\} \geq 1-\alpha$$

In [206]:
```
##STEP 0: SPECIFY PARAMETERS AND POPULATE THE SCENARIOS
alpha = 0.05
s = [20, 20, 20, 20, 40, 40, 40, 60, 60]
S = [40, 60, 80, 100, 60, 80, 100, 80, 100]
N = 100
I_ = {}
for i in range(len(ss)):
    I_[(s[i], S[i])] = simulate_inventory(s[i],S[i],N,T)
```

In [211]:
```
pd.DataFrame(I_).hist(figsize = (6,6), sharex=True, sharey=True );
```



In [212]:
```
## STEP 1 Calculate the t quantiles for each scenario
K = len(ss);
t = {};
ni = {};
q = (1 - alpha)**(1/(K-1))

for key in I_.keys():
    ni[key] = len(I_[key])
    t[key] = stats.t.ppf(q, ni[key]-1)
t
```

Out[212]:
```
{(20, 40): 2.2575896178284887,
 (20, 60): 2.2575896178284887,
 (20, 80): 2.2575896178284887,
 (20, 100): 2.2575896178284887,
 (40, 60): 2.2575896178284887,
 (40, 80): 2.2575896178284887,
 (40, 100): 2.2575896178284887,
 (60, 80): 2.2575896178284887,
 (60, 100): 2.2575896178284887}
```

In [213]:
```
## STEP 2.1: Calculate the sample means and sample variances
sample_means = {}
sample_vars = {}

for key, value in I_.items():
    sample_means[key] = np.mean(value)
    sample_vars[key] = np.var(value, ddof=1)

## STEP 2.2: Calculate the pairwise thresholds
Thresholds = {}
for key_i in I_.keys():
    for key_h in I_.keys():
        if key_i != key_h:
            #not the most computatinally efficient way.
            Thresholds[(key_i, key_h)] = ((t[key_i]**2)*sample_vars[key_i]/ni[key_i] + (t[key_h]**2)*s
ample_vars[key_h]/ni[key_h])**0.5
```

In [214]:
```
## STEP 3.0: Form the subset of scenarios that are less than the others plus the threshold between the
m
Out = set()
for key_i in I_.keys():
    flag = 1
    for key_h in I_.keys():
        if key_i != key_h and sample_means[key_i] >  sample_means[key_h] +  Thresholds[(key_i, key_h
)]:
            flag = 0
    if flag:
        Out.add(key_i)
```

In [215]:
```
Out
```

Out[215]:
```
{(20, 100)}
```

Therefore the best design is s = 20, S = 100 with 95% confidence!

Part d)

The selection of the best algoirthm guarantees that: $Pr\{\text{select } x_B|\theta(x_i)-\theta(x_b)\geq\delta, \forall i\neq B\}\geq 1-\alpha$

i.e there is over a 1-α probability that the best scenario is selected conditional on the cost difference between the best scenario and the other scenarios being larger than delta. In the case of $\delta = 10$ and $\alpha = 0.05$ and $(s, S) = (20, 60)$: it follows that out of all the scenarios that have over a 10 dollar higher average cost we are 95% confident that $(20, 60)$ is the best design.

Furthermore, if there is is a scenario that has an expected cost within 10 dollars of the selected scenario then the selected scenario $20, 60$ is within ten dollars of the best.

Problem # 3

In [ ]:
```
```