Student name: Steven Xie

Student number: 998979627

Program & Department: MEng MIE

# Homework 4

**Problem 1.**

(a).

When the simulation begins, for each replication, we set the initial inventory I = 60 and event time t = 0, and the ordering amount a = 0, then schedule the shipping event by running Shipping function and ordering event by running Ordering function.

In the Shipping function, we first call the update\_hcost function to update total cost of inventory/backlog. Then update the inventory level by the demandschedule function which uses SimRNG.Uniform(0, 1, 2) <= p to represent with probability p. Note that the actual inventory level cannot be negative, but variable I can be negative to make it easier to compute the backlog cost. Inventory level is recorded by CTStats function Inventory and event time t is updated. Then schedule the next Shipping event by Expon(0.1) as stated in the problem.

In the Ordering function, if the inventory level is less than s, and there is no active order (a = 0), we make an order of amount S – I, and schedule the receiving event of the order, and update the total cost by the ordering cost (K + a \* i). Then schedule the next Ordering event 1 month after. In the Receiving function, we first update the total cost of inventory since last event that has a change in inventory. Then we added the ordered amount to the inventory, update the event time and reset the ordering amount to 0.

(b).

By setting s = 20, S = 50, run 1000 replications.

Estimate average cost with a 95% CI is: (14607.052961511064, 14558.872717724009, 14655.23320529812)

Code shown below:

*import* SimFunctions  
*import* SimRNG  
*import* SimClasses  
*import* numpy *as* np  
*import* scipy.stats *as* stats  
  
ZSimRNG = SimRNG.InitializeRNSeed()  
Calendar = SimClasses.EventCalendar()  
  
  
*def* t\_mean\_confidence\_interval(data, alpha): *# compute the CI with set alpha* a = 1.0 \* np.array(data)  
 n = len(a)  
 m, se = np.mean(a), stats.sem(a)  
 h = stats.t.ppf(1 - alpha / 2, n - 1) \* se  
 *return* m, m - h, m + h  
  
  
SimClasses.Clock = 0  
  
Inventory = SimClasses.CTStat()  
Totalcost = 0  
  
TheCTStats = []  
TheDTStats = []  
TheQueues = []  
TheResources = []  
  
I = 60 *# update the inventory*t = 0 *# update the event time whenever there is a change in inventory*a = 0 *# update the amount of orders*TheCTStats.append(Inventory)  
AllTotalcost = []  
  
  
*# Demand function  
def* demandschedule(D):  
 *if* SimRNG.Uniform(0, 1, 2) <= 1 / 6:  
 *return* D[0]  
 *elif* SimRNG.Uniform(0, 1, 2) <= 1 / 2:  
 *return* D[1]  
 *elif* SimRNG.Uniform(0, 1, 2) <= 5 / 6:  
 *return* D[2]  
 *else*:  
 *return* D[3]  
  
  
D = [1, 2, 3, 4]  
meandt = 0.1  
meanrt = 0.75  
  
*# Cost of ordering*K = 32 *# fixed cost*i = 3 *# variable cost per item  
  
# cost of inventory*h = 1 *# per item per month*pi = 5 *# backlog cost*T = 120 *# run length  
# reorder point and order to point*s = 20  
S = 50  
  
repN = 1000 *# number of replications  
  
  
def* update\_hcost(inventory, time):  
 *global* Totalcost  
 *if* inventory >= 0:  
 Totalcost += inventory \* h \* (SimClasses.Clock - time) *# holding cost  
 else*:  
 Totalcost += abs(inventory) \* pi \* (SimClasses.Clock - time) *# backlog cost  
  
  
def* Shipping(): *# ship out or use inventory to satisfy demand  
 global* Totalcost  
 *global* I  
 *global* t  
 update\_hcost(I, t)  
 I -= demandschedule(D)  
 Inventory.Record(max(0, I))  
 t = SimClasses.Clock  
 SimFunctions.Schedule(Calendar, "Shipping", SimRNG.Expon(meandt, 2))  
 *# meet demand use inventory  
 # update inventory level and compute cost  
  
  
def* Ordering(): *# check invenotry level and make orders  
 global* Totalcost  
 *global* I  
 *global* t  
 *global* a  
 *if* I < s *and* a == 0: *# make orders and update totalcost by ordering cost* a = S - I  
 *# Schedule receiving* SimFunctions.Schedule(Calendar, "Receiving", SimRNG.Erlang(2, meanrt, 2))  
 Totalcost += K + a \* i  
 SimFunctions.Schedule(Calendar, "Ordering", 1)  
  
  
*def* Receiving():  
 *global* Totalcost  
 *global* I  
 *global* t  
 *global* a  
 update\_hcost(I, t)  
 I += a *# update inventory level* Inventory.Record(max(0, I))  
 t = SimClasses.Clock  
 a = 0 *# reset order amount  
  
  
for* reps *in* range(0, repN, 1):  
 Totalcost = 0 *# Reset the cost for each replication* I = 60  
 t = 0  
 a = 0  
 SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats, TheDTStats, TheResources)  
 SimFunctions.Schedule(Calendar, "Shipping", SimRNG.Expon(meandt, 2))  
 SimFunctions.Schedule(Calendar, "Ordering", 1)  
 SimFunctions.Schedule(Calendar, "EndSimulation", T)  
  
 NextEvent = Calendar.Remove()  
 SimClasses.Clock = NextEvent.EventTime  
 *if* NextEvent.EventType == "Shipping":  
 Shipping()  
 *elif* NextEvent.EventType == "Ordering":  
 Ordering()  
 *elif* NextEvent.EventType == "Receiving":  
 Receiving()  
  
 *while* NextEvent.EventType != "EndSimulation":  
 NextEvent = Calendar.Remove()  
 SimClasses.Clock = NextEvent.EventTime  
 *if* NextEvent.EventType == "Shipping":  
 Shipping()  
 *elif* NextEvent.EventType == "Ordering":  
 Ordering()  
 *elif* NextEvent.EventType == "Receiving":  
 Receiving()  
  
 AllTotalcost.append(Totalcost)  
 *# print(Inventory.Mean())*print("Estimate average cost with a 95% CI is: ", t\_mean\_confidence\_interval(AllTotalcost, 0.05))

Run result:



(c).

To implement CRN, we will run separately for each scenario of (s, S) policy and record the outputs mean and variance below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Scenario | X1 | X2 | X3 | X4 | X5 | X6 |
| s | 20 | 20 | 20 | 30 | 30 | 30 |
| S | 50 | 60 | 70 | 60 | 80 | 100 |
| Replication | 100 | 100 | 100 | 100 | 100 | 100 |
| Mean(Yi) | 14563.12 | 14120.54 | 14183.89 | 14193.99 | 14392.32 | 15070.28 |
| Threshold | 14330.00 | 14286.95 | 14286.85 | 14290.50 | 14277.63 | 14264.39 |
| Variance(Si) | 540718.99 | 249321.93 | 248723.14 | 270852.59 | 195064.12 | 123330.97 |
|  |  |  |  |  |  |  |
| Replication | 500 | 500 | 500 | 500 | 500 | 500 |
| Mean(Yi) | 14609.78 | 14181.33 | 14194.31 | 14234.95 | 14412.74 | 15081.34 |
| Threshold | 14413.26 | 14385.44 | 14372.63 | 14373.44 | 14354.75 | 14349.70 |
| Variance(Si) | 609189.88 | 384984.86 | 291411.53 | 297170.89 | 170901.57 | 138963.18 |

For n = 100 replications, we simulate K = 6 scenarios for different (s, S) policies. X2 has the smallest sample average. We calculate a threshold for subset selection with equation:

Threshold = Y2 + sqrt(ti2 \* Si / 100 + t22 \* Si/100),

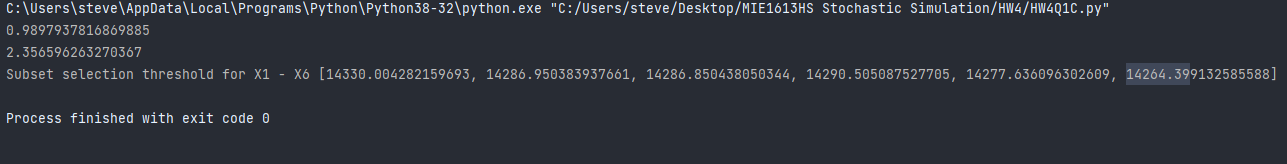
If Xi ≤ Threshold, the policy is selected, otherwise eliminated.

As a result, the subset {X2,X3,X4} is selected as good policies.

Calculation code shown below:

*import* scipy.stats  
*import* numpy *as* np  
  
q = 0.95 \*\* (1 / 5)  
print(q)  
*# find T critical value*t = scipy.stats.t.ppf(q=q, df=99)  
print(t)  
  
X = [14563.12, 14120.54, 14183.89, 14193.99, 14392.32, 15070.28]  
S = [540718.99, 249321.93, 248723.14, 270852.59, 195064.12, 123330.97]  
Y = []  
*for* i *in* range(6):  
 threshold = X[1] + np.sqrt(t \* t \* S[i]/100 + t \* t \* S[1]/100)  
 Y.append(threshold)  
print('Subset selection threshold for X1 - X6', Y)

Result for 100 reps:



For n = 500 replications, we simulate K = 6 scenarios for different (s, S) policies. X2 has the smallest sample average. We calculate a threshold for subset selection with equation:

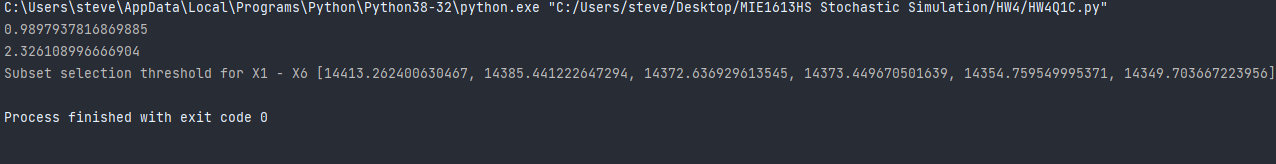
Threshold = Y2 + sqrt(ti2 \* Si / 500 + t22 \* Si/500),

If Xi ≤ Threshold, the policy is selected, otherwise eliminated.

As a result, the subset {X2,X3,X4} is selected as good policies.

Calculation code:

*import* scipy.stats  
*import* numpy *as* np  
  
q = 0.95 \*\* (1 / 5)  
print(q)  
*# find T critical value  
#t = scipy.stats.t.ppf(q=q, df=99)*t = scipy.stats.t.ppf(q=q, df=499)  
print(t)  
  
*# X = [14563.12, 14120.54, 14183.89, 14193.99, 14392.32, 15070.28]  
# S = [540718.99, 249321.93, 248723.14, 270852.59, 195064.12, 123330.97]  
# Y = []*X500 = [14609.78, 14181.33, 14194.31, 14234.95, 14412.74, 15081.34]  
S500 = [609189.88, 384984.86, 291411.53, 297170.89, 170901.57, 138963.18]  
Y500 = []  
*for* i *in* range(6):  
 threshold = X500[1] + np.sqrt(t \* t \* S500[i] / 100 + t \* t \* S500[1] / 100)  
 Y500.append(threshold)  
print('Subset selection threshold for X1 - X6', Y500)

Result for 500 reps: 

Problem 2.

(a).

The simulation model is built on the model SAN\_Max\_CRN.py. Set s1 to be the initial x = (0.5, 1, 0.7, 1, 1),X1 Y1 is the Y(x) which is the same as the original model. X2 Y2 is used to simulate Y(X+△X) for each x(d). The gradient is computed by (Y(X+△X) – Y(X))/ △X for each X(i). Run 1000 replications and gradient of E(Y(x)) and 95% confidence interval is computed by adapted CI\_95 function. Note that CRN is used for each replication.

From the results below, we can see that as △X decreases, the upper and lower of confidence interval gets wider which indicates higher variance, but the bias of estimate is reduced for smaller △X, which illustrated the bias-variance tradeoff.

For △X = 0.1:

Expected gradient is [0.0, -0.6172120770068538, -0.6071567154734367, -0.4556849387779693, -0.883270017889545]

Upper bound is [0.0, -0.5506038420147474, -0.5433442189928716, -0.3914382281119872, -0.8197828043489994]

Lower bound is [0.0, -0.6838203119989601, -0.6709692119540019, -0.5199316494439513, -0.9467572314300905]

For △X = 0.05:

Expected gradient is [0.0, -0.6287100953263707, -0.6174950758625857, -0.46068334663255855, -0.886357200524703]

Upper bound is [0.0, -0.5618134663439519, -0.5530207998038118, -0.3962832410084045, -0.8228878512374075]

Lower bound is [0.0, -0.6956067243087894, -0.6819693519213595, -0.5250834522567126, -0.9498265498119985]

For △X = 0.01:

Expected gradient is [0.0, -0.6416801500232943, -0.6250737762807144, -0.4667341429138154, -0.8883624897080078]

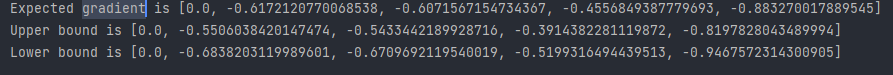
Upper bound is [0.0, -0.5744058637305351, -0.5603502039550246, -0.4020830957818783, -0.8249289167329936]

Lower bound is [0.0, -0.7089544363160534, -0.6897973486064042, -0.5313851900457525, -0.951796062683022]

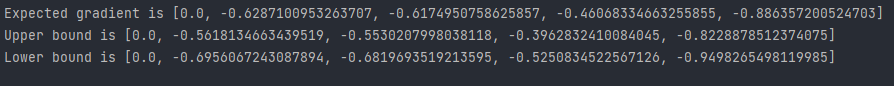
Code shown below:

*import* numpy *as* np  
*from* copy *import* copy  
  
  
*def* CI\_95(data): *# compute the 95% confidence interval for columns n \* m np array* n, m = data.shape  
 a = np.mean(data, axis=0)  
 sd = np.std(data, axis=0)  
 hw = 1.96 \* sd / np.sqrt(n)  
 print("Expected gradient is", a.tolist())  
 print("Upper bound is", (a+hw).tolist())  
 print("Lower bound is", (a-hw).tolist())  
  
  
np.random.seed(1)  
  
N = 1000 *# number of replications*s1 = [0.5, 1, 0.7, 1, 1] *# initial X*deltaX = 0.01 *# 0.1 0.05 0.01*FD = []  
  
*for* rep *in* range(0, N, 1):  
 U = np.random.random(5)  
 X1 = []  
 X2 = []  
 *for* i *in* range(0, 5, 1):  
 X1.append(-np.log(1 - U[i]) \* s1[i])  
  
 *for* i *in* range(5):  
 X2 = copy(X1)  
 X2[i] = (-np.log(1 - U[i]) \* max(0.5, s1[i] - deltaX))  
 Y1 = max(X1[0] + X1[3], X1[0] + X1[2] + X1[4], X1[1] + X1[4]) *# Y(X)* Y2 = max(X2[0] + X2[3], X2[0] + X2[2] + X2[4], X2[1] + X2[4]) *# Y(X + deltaX)* FD.append((Y2 - Y1) / deltaX) *# Compute the FD*A = np.array(FD) *# convert the list to np array to calculate Mean and CI for all FD*A = A.reshape((N, 5))  
*# print(A)  
# print(np.mean(A, axis=0))  
# print(np.std(A, axis=0))*CI\_95(A)

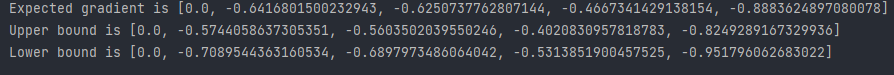
Result for deltaX =0.1:



Result for deltaX = 0.05:



Result for deltaX = 0.01:



(b).

Gradient = dy/dx(i) = - ln (1 – U(i)) if x(i) is on the longest path,

= 0 O/W

To check if x(i) is on the longest path, we make X2[i] = (X(i) + △X) where △X is a extreme small number(i.e 0.00001), if Y1 = Y2, then the X(i) is not on the longest path, so gradient using IPA = 0, otherwise X(i) is on the longest path so gradient = - ln (1 – U(i)). Run 1000 replications and compute the 95% confidence interval.

Result:

Expected gradient is [0.7388930164995414, 0.6423846633968452, 0.627503639503355, 0.467690557942007, 0.8886716623413301]

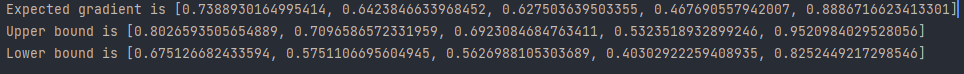
Upper bound is [0.8026593505654889, 0.7096586572331959, 0.6923084684763411, 0.5323518932899246, 0.9520984029528056]

Lower bound is [0.675126682433594, 0.5751106695604945, 0.5626988105303689, 0.40302922259408935, 0.8252449217298546]

Code shown below:

*import* numpy *as* np  
*from* copy *import* copy  
  
  
*def* CI\_95(data): *# compute the 95% confidence interval for columns n \* m np array* n, m = data.shape  
 a = np.mean(data, axis=0)  
 sd = np.std(data, axis=0)  
 hw = 1.96 \* sd / np.sqrt(n)  
 print("Expected gradient is", a.tolist())  
 print("Upper bound is", (a + hw).tolist())  
 print("Lower bound is", (a - hw).tolist())  
  
  
np.random.seed(1)  
  
N = 1000 *# number of replications*s1 = [0.5, 1, 0.7, 1, 1] *# initial X*IPA = []  
  
*for* rep *in* range(0, N, 1):  
 U = np.random.random(5)  
 X1 = []  
 X2 = []  
 *for* i *in* range(0, 5, 1):  
 X1.append(-np.log(1 - U[i]) \* s1[i])  
  
 *for* i *in* range(5):  
 X2 = copy(X1)  
 gradX = (-np.log(1 - U[i])) *# compute the gradient of IPA* X2[i] = X1[i] + 0.00001  
 Y1 = max(X1[0] + X1[3], X1[0] + X1[2] + X1[4], X1[1] + X1[4]) *# Y(X)* Y2 = max(X2[0] + X2[3], X2[0] + X2[2] + X2[4], X2[1] + X2[4]) *# Y(X + deltaX)  
 if* Y1 == Y2: *# X(i) is not on the longest path* IPA.append(0)  
 *else*: *# X(i) is on the longest path* IPA.append(gradX)  
A = np.array(IPA) *# convert the list to np array to calculate Mean and CI for all FD*A = A.reshape((N, 5))  
*# print(A)  
# print(np.mean(A, axis=0))  
# print(np.std(A, axis=0))*CI\_95(A)

Result shown below:



(c).

The approach is to use IPA to find the gradient and improve the x for 100 times with x[i+1] = x[i] + gradient \* r. We picked r = 1/ (N + 500) as a learning rate for stochastic approximation.

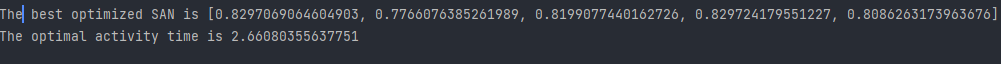
We find the best optimized SAN is [0.8297069064604903, 0.7766076385261989, 0.8199077440162726, 0.829724179551227, 0.8086263173963676]

The optimal activity time is 2.66080355637751 for 100 replications.

Code shown below:

*import* numpy *as* np  
*from* copy *import* copy  
  
  
*def* CI\_95(data): *# compute the 95% confidence interval for columns n \* m np array* n, m = data.shape  
 a = np.mean(data, axis=0)  
 sd = np.std(data, axis=0)  
 hw = 1.96 \* sd / np.sqrt(n)  
 print("Expected gradient is", a.tolist())  
 print("Upper bound is", (a + hw).tolist())  
 print("Lower bound is", (a - hw).tolist())  
  
  
np.random.seed(1)  
  
N = 100 *# number of gradient descent applied*s1 = [1, 1, 1, 1, 1] *# initial X*S = []  
Y = []  
X2 = [1,1,1,1,1]  
*for* rep *in* range(0, N, 1):  
 U = np.random.random(5)  
 r = 1/(rep+500) *# set r that's going to 0 but sum to infinity* X1 = []  
 *for* i *in* range(0, 5, 1):  
 X1.append(-np.log(1 - U[i]) \* s1[i])  
 *for* i *in* range(5):  
 X2 = copy(X1)  
 gradX = (-np.log(1 - U[i])) *# compute the gradient of IPA* s1[i] = max(s1[i]-gradX \* r,0.5)  
 X2[i] = (-np.log(1 - U[i]) \* max(0.5, s1[i] - gradX))  
 S.append(s1)  
 Y.append(max(X2[0] + X2[3], X2[0] + X2[2] + X2[4], X2[1] + X2[4])) *# Y(X + deltaX)*print("The best optimized SAN is", S[Y.index(min(Y))])  
yfinal = []  
*for* rep *in* range(100):  
 U = np.random.random(5)  
 Sfinal = S[Y.index(min(Y))]  
 X3 = []  
 *for* i *in* range(0, 5, 1):  
 X3.append(-np.log(1 - U[i]) \* s1[i])  
 yfinal.append(max(X3[0] + X3[3], X3[0] + X3[2] + X3[4], X3[1] + X3[4]))  
print('The optimal activity time is', np.mean(yfinal))

Result shown below:



Problem 3.

(a).

When x > 5, F-1(x) =

When x < 5, F-1(x) can be any constant in [0,5].

So the inversion algorithm is:

Generate U ~ Unif[0,1]

if x[i] <5, Return D = 5U

If X[i] ≥ 5, Return D =

(b).

dθ(x)/dx = d/dx E[c0 max(x − D, 0) + cu max(D − x, 0)]

≈E[d/dx (c0 max(x − D, 0) + cu max(D − x, 0))]

= E(c0 if x ≥ D, -cu if x < D)

= E(c0 if x ≥, -cu if x <)

Generate n iid samples of Unif[0,1]: U1,U2….Un and use:

(c).

The output of θ(x) can be express as yi(x) = c0 max(x − , 0) + cu max( − x, 0) if x ≥ 5

= c0 max(x – 5U, 0) + cu max( − x, 0) if x < 5

Object: Min (x)

ST: ≥ c0 (x -

≥ cu (

≥ c0 (x - 5U)

yi ≥ cu (5U – x)

0 ≤ x ≤ 50