Stochastic Simulation (MIE1613H) - Homework 3 (Solutions)

- Submit your homework on Quercus as a single PDF file by the deadline. Late submissions are penalized 10% each day the homework is late.
- At the top of your homework include your name, student number, department, and program.
- You may discuss the assignment with other students in general terms, but each student must solve the problems, write the code and the solutions individually.
- **The simulation models must be programmed in Python. For the input modelling questions you may use R or Python.** You must include both the source code (including comments to make it easy to follow) and the output of the simulation in your submission.
- **It is very important to explain your answers and solution approach clearly and not just provide the code/results**. Full marks will only be given to correct solutions that are fully and clearly explained.

Problem 1. (25 Pts.) You are asked to make projections about cycle times for a semiconductor manufacturer who plans to open a new plant. Here "cycle time" means the time from product release until completion. The process that you will consider has a single diffusion step with sub-steps as indicated in the diagram.

Raw material for two products (C) and (D) will begin at the CLEAN step, and make multiple passes until the product completes processing. Movement within each process is handled by robots and takes very little time (treat as 0) relative to the processing steps. The movement time from release to diffusion is 15 minutes (1/4 hour).

The anticipated product mix is to have 60% of products to be of type (C) and 40% of type (D). Product (C) requires 5 passes and product D requires 3 passes.

The OXDIZE step is deterministic but differs by product type: it takes 2.7 hours for (C) and 2.0 hours for (D).

(a) The CLEAN and LOAD/UNLOAD steps do not differ by product type but are subject to uncertainty. Historical data is provided in (SemiconductorData.xls). Represent them using the distributions available in PythonSim, and justify your choice using the graphical and statistical methods discussed in the lecture.

Product will be released in cassettes at the rate of 1 cassette/hour, 7 days a week, 24-hours a day (this is to achieve a desired throughput of 1 cassette/hour). Product is moved and processed in single cassette loads.

The table below shows the number of machines that are planned for each fabrication step:

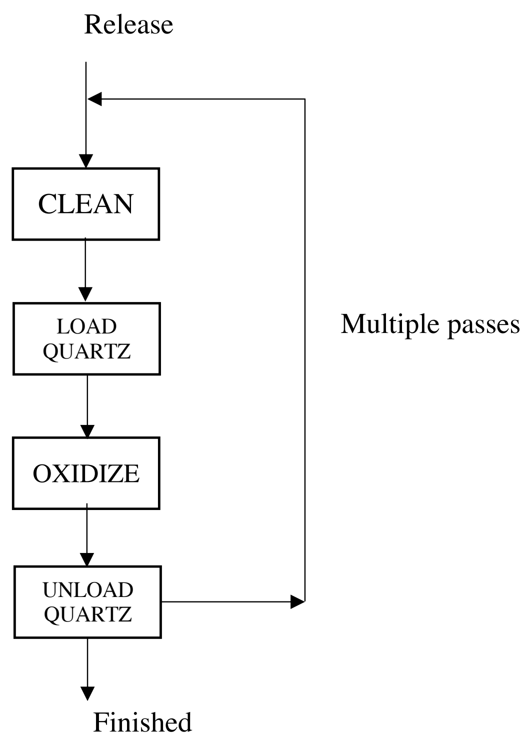

Figure 1: Diagram of the diffusion step.

Step	**Number of Machines**
CLEAN	9
LOAD QTZ	2
OXD	11
UNLOAD QTZ	2

Table 1: Number of available machines in each step.

(b) Provide estimates of the long-run average cycle times for each product. Determine and justify a warm-up period and run length. You may use the replication-deletion approach. Provide appropriate confidence intervals for your estimates.

(a) **(10 Points)** Since we do not have a physical justification for the choice of distributions, we can speculate a few candidate distributions (e.g., by looking at the shape of the histogram) in each case and examine the goodness of fit for them. For the CLEAN data, we pick Exponential, Weibull, and Gamma distributions. The KS test does not reject any of the distributions at $(1-\alpha) = 95\%$ confidence level. Examining the Q-Q plot we observe that the fits are very similar, so any of them would provide an acceptable choice. We pick exponential that is available in PythonSim. The estimated parameter for exponential is $\lambda = 0.5512835$, so the estimate of mean, that is used in PythonSim, is $1/0.5512835 = 1.813949$.

For the LOAD/UNLOAD data, we pick Lognormal, Weibull, and Gamma. The KS test does not reject any of the distributions. Examining the Q-Q plot however suggests that Gamma provides a better fit at the tail. However, we pick Lognormal that is available in PythonSim. Let X be the

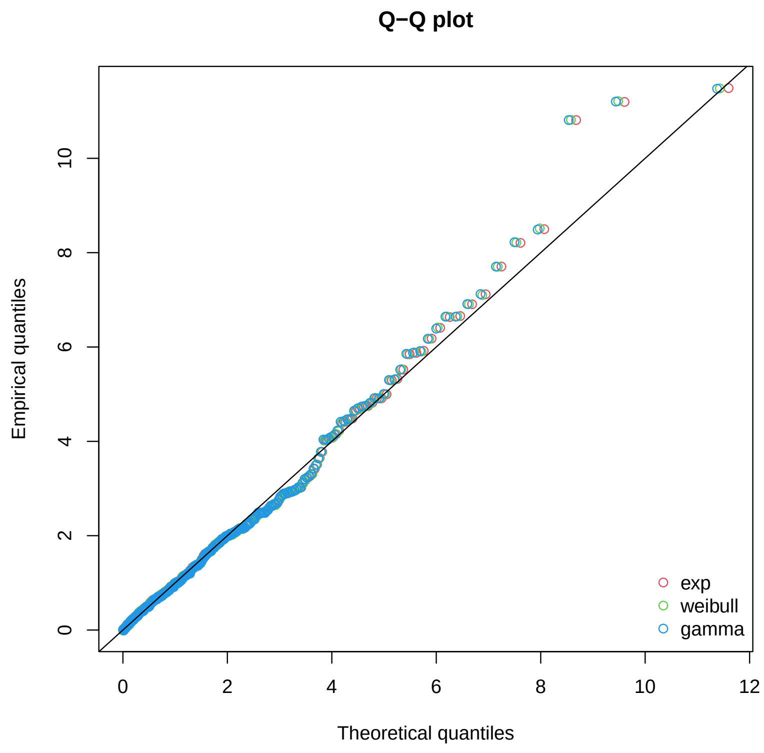

Figure 2: Q-Q plot for the fitted distributions to the CLEAN data.

Lognormal random variable and denote by μ and σ the mean and standard deviation of log(X) as estimated in R. The mean and standard deviation of X, that is required in PythonSim, are given by,

$$\mathrm{E}(X) = e^{\mu+\frac{\sigma^2}{2}} = 0.2475798,$$

$$\mathrm{SD}(X) = e^{\mu+\frac{\sigma^2}{2}}\sqrt{e^{\sigma^2}-1} = 0.2233498.$$

```
# This file uses the fitdistrplus package. You'll need to install it first
# by running the command: 'install.packages("fitdistrplus")' in R
require(fitdistrplus)

# import the data from .csv file
MyData <- read.csv("SemiconductorData.csv", header=FALSE)
Load <- MyData[,2]
Clean <- MyData[1:299,4]
plotdist(Load, histo = TRUE, demp = TRUE)
plotdist(Clean, histo = TRUE, demp = TRUE)

# fit a lognormal,gamma, and weibull to load data
load_fln <- fitdist(Load, "lnorm")
load_fw <- fitdist(Load, "weibull")
load_fg <- fitdist(Load, "gamma")

```

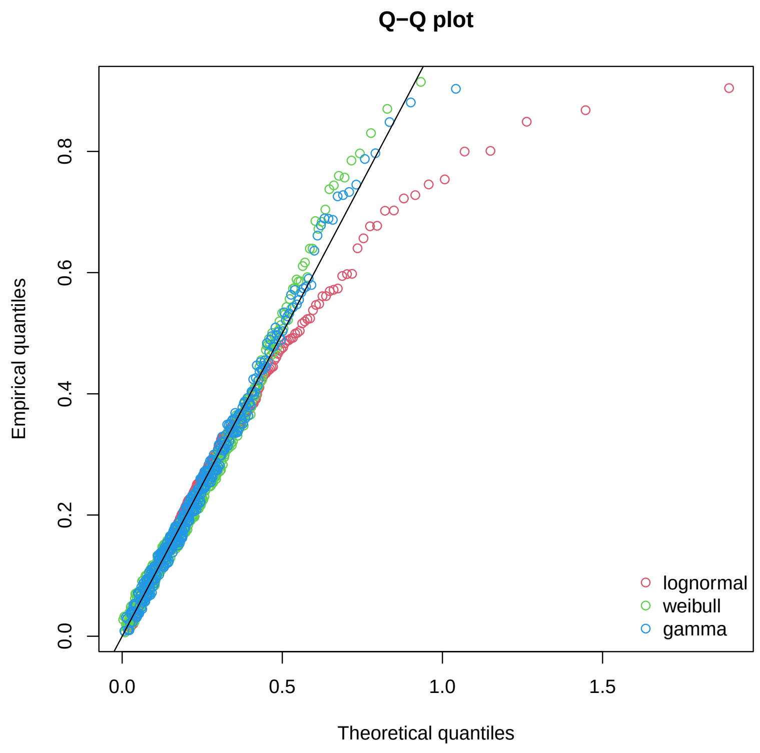

Figure 3: Q-Q plot for the fitted distributions to the LOAD data.

```
# fit an exponential, gamma, and weibull to clean data
clean_fe <- fitdist(Clean, "exp")
clean_fw <- fitdist(Clean, "weibull")
clean_fg <- fitdist(Clean, "gamma")

# compare the fits for load
plot.legend <- c("lognormal", "weibull", "gamma")
qqcomp(list(load_fln, load_fw, load_fg), legendtext = plot.legend)

# compare the fits for clean
plot.legend <- c("exp", "weibull", "gamma")
qqcomp(list(clean_fe, clean_fw, clean_fg), legendtext = plot.legend)

# Perform GofFit tests
gof_load <- gofstat(list(load_fln, load_fw, load_fg), fitnames = c("lognormal"
    , "weibull", "gamma"))
gof_load
gof_load$kstest

gof_clean <- gofstat(list(clean_fe, clean_fw, clean_fg), fitnames = c("exp", "
    weibull", "gamma"))
gof_clean
gof_clean$kstest
```

(b) **(15 Points)** We define a special entity class in SimClasses.py as follows to represent a product.

```python
# special entity to represent a product
class Product():
    def __init__(self, type):
        self.CreateTime = Clock
        self.Type = type
        if self.Type == "C":
            self.Passes = 5
            self.Oxd = 2.7
        else:
            self.Passes = 3
            self.Oxd = 2.0
```

When a product is released, we schedule the next release for the next hour, determine the product type with the given probabilities, and schedule the end of movement from release to diffusion using the SchedulePlus function. In the EndMove function, we schedule the end of cleaning if a cleaning machine is available using the estimated exponential distribution from the data. Otherwise, we put the product in the cleaning queue. In the EndClean function, we schedule the end of loading if a loading machine is available using the estimated Lognormal distribution from the data. Otherwise, we put the product in the loading queue. We also check the cleaning queue and if there is a product waiting, we schedule the next end of cleaning for that product. Otherwise, we make a cleaning machine idle. The EndLoad and EndOxd functions are also similar to the EndClean function. In the EndUnload function, we keep track of the product passes and if a product reaches the required passes, we record the cycle time for that product type in its corresponding DTStat object. Otherwise, we check the status of the cleaning machines and repeat the same steps.

Fig. 4 shows the mean cycle time by product number for the first $m = 1000$ products of each type averaged across $n = 30$ replications. At somewhere around 100 products the average cycle times seem to vary around a central value, so we may take $d = 100$ and choose the run length at least $m = 1100$ of each product. The estimate of the average cycle times for product C and D is 31.296 and 16.700 hours with the 95% CI of [31.28, 31.31] and [16.69, 16.71], respectively.

```python
# Cycle times for a semiconductor manufacturer
import SimFunctions
import SimRNG
import SimClasses
import scipy.stats as sp
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

def mean_confidence_interval(data, confidence=0.95):
    a = 1.0*np.array(data)
    n = len(a)
    m, se = np.mean(a), sp.sem(a)
    h = se * sp.t._ppf((1+confidence)/2., n-1)
    return m, m-h, m+h

ZSimRNG = SimRNG.InitializeRNSeed()
Calendar = SimClasses.EventCalendar()

# set up queues
```

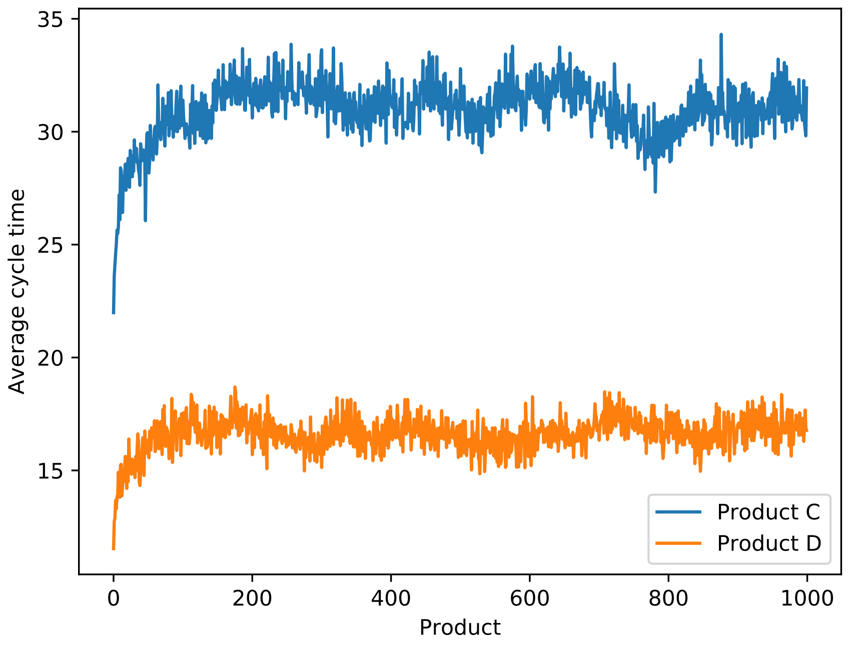

Figure 4: Mean plot, by product number, averaged across $n = 30$ replications.

```
CleanQ = SimClasses.FIFOQueue()
LoadQ = SimClasses.FIFOQueue()
OxdQ = SimClasses.FIFOQueue()
UnloadQ = SimClasses.FIFOQueue()
# set up resources
Clean = SimClasses.Resource()
Load = SimClasses.Resource()
Oxd = SimClasses.Resource()
Unload = SimClasses.Resource()

Clean.SetUnits(9)
Load.SetUnits(2)
Oxd.SetUnits(11)
Unload.SetUnits(2)

# set up statistics
CycleTimeC = SimClasses.DTStat()
CycleTimeD = SimClasses.DTStat()

TheCTStats = []
TheDTStats = [CycleTimeC, CycleTimeD]
TheQueues = [CleanQ, LoadQ, OxdQ, UnloadQ]
TheResources = [Clean, Load, Oxd, Unload]


def Release():
    SimFunctions.Schedule(Calendar, "Release", 1.0)
    if SimRNG.Uniform(0, 1, 1) < 0.6:
        NewProduct = SimClasses.Product("C")
    else:
        NewProduct = SimClasses.Product("D")

    SimFunctions.SchedulePlus(Calendar, "EndMove", 0.25, NewProduct)

def EndMove(TheProduct):
    if Clean.Busy < Clean.NumberOfUnits:
        Clean.Seize(1)
        SimFunctions.SchedulePlus(Calendar, "EndClean", SimRNG.Expon(1.813949,
             2), TheProduct)
    else:
        CleanQ.Add(TheProduct)


def EndClean(TheProduct):
    if Load.Busy < Load.NumberOfUnits:
        Load.Seize(1)
        SimFunctions.SchedulePlus(Calendar, "EndLoad", SimRNG.Lognormal
            (0.2475798, 0.2233498 ** 2, 3), TheProduct)
    else:
        LoadQ.Add(TheProduct)

    if CleanQ.NumQueue() > 0:
        NextProduct = CleanQ.Remove()
        SimFunctions.SchedulePlus(Calendar, "EndClean", SimRNG.Expon(1.813949,
             2), NextProduct)
    else:
        Clean.Free(1)


def EndLoad(TheProduct):
    if Oxd.Busy < Oxd.NumberOfUnits:
        Oxd.Seize(1)
        SimFunctions.SchedulePlus(Calendar, "EndOxd", TheProduct.Oxd,
            TheProduct)
    else:
        OxdQ.Add(TheProduct)

    if LoadQ.NumQueue() > 0:
        NextProduct = LoadQ.Remove()
        SimFunctions.SchedulePlus(Calendar, "EndLoad", SimRNG.Lognormal
            (0.2475798, 0.2233498 ** 2, 3), NextProduct)
    else:
        Load.Free(1)


def EndOxd(TheProduct):
    if Unload.Busy < Unload.NumberOfUnits:
        Unload.Seize(1)
        SimFunctions.SchedulePlus(Calendar, "EndUnload", SimRNG.Lognormal
            (0.2475798, 0.2233498 ** 2, 3), TheProduct)
    else:
```

```python
        UnloadQ.Add(TheProduct)

    if OxdQ.NumQueue() > 0:
        NextProduct = OxdQ.Remove()
        SimFunctions.SchedulePlus(Calendar, "EndOxd", NextProduct.Oxd, NextProduct)
    else:
        Oxd.Free(1)


def EndUnload(TheProduct):
    global CTC, CTD, TC, TD
    TheProduct.Passes = TheProduct.Passes - 1
    if TheProduct.Passes > 0:
        if Clean.Busy < Clean.NumberOfUnits:
            Clean.Seize(1)
            SimFunctions.SchedulePlus(Calendar, "EndClean", SimRNG.Expon(1.813949, 2), TheProduct)
        else:
            CleanQ.Add(TheProduct)
    else:
        if TheProduct.Type == "C":
            CycleTimeC.Record(SimClasses.Clock - TheProduct.CreateTime)
            CTC.append(SimClasses.Clock - TheProduct.CreateTime)
            TC += 1
        else:
            CycleTimeD.Record(SimClasses.Clock - TheProduct.CreateTime)
            CTD.append(SimClasses.Clock - TheProduct.CreateTime)
            TD += 1

    if UnloadQ.NumQueue() > 0:
        NextProduct = UnloadQ.Remove()
        SimFunctions.SchedulePlus(Calendar, "EndUnload", SimRNG.Lognormal(0.2475798, 0.2233498 ** 2, 3), NextProduct)
    else:
        Unload.Free(1)

AcrossC = []
AcrossD = []
AllCTC = [] # these will be a list of lists, each
AllCTD = [] # list corresponding to one replication's cycle times

m = 1100 #
d = 100 # Deletion point
for reps in range(0, 30, 1):
    CTC = [] # cycle times from current replication
    CTD = [] # cycle times from current replication
    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats, TheDTStats, TheResources)
    SimFunctions.Schedule(Calendar, "Release", 1.0)

    NextEvent = Calendar.Remove()
    SimClasses.Clock = NextEvent.EventTime
    Release()

    TC = 0 # Number of completed products of type C
    TD = 0 # Number of completed products of type D
    while min(TC, TD) < m:
        if min(TC, TD) == d:
            SimFunctions.ClearStats(TheCTStats, TheDTStats)

        NextEvent = Calendar.Remove()
        SimClasses.Clock = NextEvent.EventTime
        if NextEvent.EventType == "Release":
            Release()
        elif NextEvent.EventType == "EndMove":
            EndMove(NextEvent.WhichObject)
        elif NextEvent.EventType == "EndClean":
            EndClean(NextEvent.WhichObject)
        elif NextEvent.EventType == "EndLoad":
            EndLoad(NextEvent.WhichObject)
        elif NextEvent.EventType == "EndOxd":
            EndOxd(NextEvent.WhichObject)
        elif NextEvent.EventType == "EndUnload":
            EndUnload(NextEvent.WhichObject)


    AcrossC.append(CycleTimeC.Mean())
    AcrossD.append(CycleTimeD.Mean())
    AllCTC.append(CTC)
    AllCTD.append(CTD)

# output results
print("Estimated expected cycle time for product C:", mean_confidence_interval(AcrossC, 0.05))
print("Estimated expected cycle time for product D:", mean_confidence_interval(AcrossD, 0.05))

# Create mean plot to determine the warm-up period
MeanC = []
MeanD = []
for i in range(1000): # Average cycle time for the first 1000 products
    MeanC.append(np.mean([rep[i] for rep in AllCTC]))
    MeanD.append(np.mean([rep[i] for rep in AllCTD]))
plt.plot(MeanC, label = "Product C")
plt.plot(MeanD, label = "Product D")
plt.xlabel("Product")
plt.ylabel("Average cycle time")
plt.legend()
plt.show()
```

```
Estimated expected cycle time for product C: (31.29, 31.28, 31.31)
Estimated expected cycle time for product D: (16.70, 16.69, 16.71)
```

Problem 2. (20 Pts.) An Automated external defibrillator (AED) can save a person's life in event of a cardiac arrest. To accelerate delivery, start ups have been developing drone technology to quickly deliver AEDs to the scene of the cardiac arrest in case of an emergency call. The AEDs will be maintained at various bases across the city to respond to calls in the area designated to each base.

You have been tasked to estimate the minimum number of drones at a certain base to ensure that with 95% probability there will be a drone available at the event of a cardiac arrest in the area covered by that base.

Assume that calls reporting cardiac arrests arrive according to a non-homogeneous Poisson process with (per minute) rate function,

$$\lambda(t) = \begin{cases} 4, & \text{7AM-12PM}, \\ 2, & \text{12PM-12AM}, \\ 1, & \text{12AM-7AM}. \end{cases}$$

throughout a day. Further, the total "drone busy time" i.e. from the time the call is received until the drone is back to the base and ready to dispatch again is exponentially distributed with mean 45 minutes. Assume at time 12AM, there are no calls (requests for AEDs) in the system.

(a) Explain, in words, your approach in determining the minimum number of drones at the base required to satisfy the provided service level. Specify the queueing model, corresponding parameters, and the performance measure you are estimating.

(b) Determine the minimum number of drones at the base using the inputs provided. Provide a 95% CI for your estimate.

(a) **(5 Points)** The approach is the same as the parking lot example and the $M(t)/M/\infty$ queue. The piecewise-constant arrival rate function $\lambda(t)$ can be used together with the thinning method to generate the NSPP. Note that the inter-arrival times are no longer exponentially distributed and hence generating exponential inter-arrival times with time-dependent rate is NOT a valid approach. The service times are i.i.d exponential with mean 45 minutes. The performance measure of interest is the 0.95th quantile of the maximum number of calls reporting cardiac arrests during the day.

(b) **(15 Points)** We simulate 1000 samples of the maximum number of calls during the day, sort them, and pick the 950^{th} value as the minimum number of drones required at the base for the desired service level. The estimated required drones is 220 with the 95% CI of $[Y_{(936)}, Y_{(964)}] = [218, 221]$.

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

ZSimRNG = SimRNG.InitializeRNSeed()
Calendar = SimClasses.EventCalendar()
TheCTStats = []
TheDTStats = []
TheQueues = []
TheResources = []
```

```python
AllMaxQueue = []

MeanBusyTime = 45.0
RepNum = 1000

ArrivalRates = [1, 4, 2]
MaxRate = 4

# Picewise-constant arrival rate function
def PW_ArrRate(t):
    if t < 7*60:    # 12AM-7AM
        return ArrivalRates[0]
    elif t < 12*60:    # 7AM-12PM
        return ArrivalRates[1]
    else:    # 12PM-12AM
        return ArrivalRates[2]

# Thinning method used to generate NSPP with the piecewise-constant arrival
    rate
def NSPP(Stream):
    PossibleArrival = SimClasses.Clock + SimRNG.Expon(1/MaxRate, Stream)
    while SimRNG.Uniform(0, 1, Stream) >= PW_ArrRate(PossibleArrival)/MaxRate:
        PossibleArrival = PossibleArrival + SimRNG.Expon(1/MaxRate, Stream)
    nspp = PossibleArrival - SimClasses.Clock
    return nspp

def Arrival():
    global MaxQueue
    global N
    interarrival = NSPP(1)
    SimFunctions.Schedule(Calendar, "Arrival", interarrival)
    N = N + 1
    if N > MaxQueue:
        MaxQueue = N
    SimFunctions.Schedule(Calendar, "EndOfService", SimRNG.Expon(MeanBusyTime,
         2))

def EndOfService():
    global N
    N = N - 1

for Reps in range(0, RepNum, 1):
    N = 0
    MaxQueue = 0

    SimFunctions.SimFunctionsInit(Calendar, TheQueues, TheCTStats, TheDTStats,
         TheResources)
    interarrival = NSPP(1)
    SimFunctions.Schedule(Calendar, "Arrival", interarrival)
    SimFunctions.Schedule(Calendar, "EndSimulation", 24*60)

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

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

    AllMaxQueue.append(MaxQueue)

# estimating the 0.95 quantile
print("Estimated required drones is:", np.sort(AllMaxQueue)[950])
l = int(np.floor(950 - 1.96*np.sqrt(1000*0.95*0.05)))
u = int(np.ceil(950 + 1.96*np.sqrt(1000*0.95*0.05)))
print("The 95% CI: [{}, {}]".format(np.sort(AllMaxQueue)[l], np.sort(
    AllMaxQueue)[u]))
```

```
Estimated required drones is:  220
The 95% CI: [218, 221]
```

Problem 3. (20 Pts.) The attached file (TSLA.csv) provides daily prices for Tesla's stock from March 2022 - March 2023.

(a) Fit a Geometric Brownian Motion (GBM) to the **Close** price by estimating the drift μ and volatility terms σ. Recall that for GBM the log-returns are independent Normal random variables, i.e.,

$$\log\left(\frac{S(t_i)}{S(t_{i-1})}\right) \sim N(\mu(t_i - t_{i-1}), \sigma(t_i - t_{i-1})),$$

for all $t_i, i = 1, \ldots, k$.

(b) Does the GBM fit the data well? Use your estimates from part (a) together with a Q-Q plot and statistical goodness of fit tests discussed in class to support your answer.

(a) **(12 Points)** We first calculate the log-returns using the **Close** price data. We then fit a normal distribution to the log-returns data and estimate $\mu = -0.001518507$ and $\sigma = 0.042617534$.

```
# This file uses the fitdistrplus package. You'll need to install it first
# by running the command: 'install.packages("fitdistrplus")' in R
require(fitdistrplus)

# import the data from .csv file
Data <- read.csv("TSLA.csv", header=TRUE,
                 sep=",", na.strings=c("NA", ""), stringsAsFactors=FALSE, as.is=TRUE)
Data$LogReturn = 0
for (i in 2:nrow(Data)){
  Data$LogReturn[i] = log(Data$Close[i]/Data$Close[i-1])
}

```

```
MyData <- Data$LogReturn

# fit a normal distribution to the log-returns
fln <- fitdist(MyData, "norm")
fln$estimate


# Graphical method
par(mfrow = c(1, 2))
plot.legend <- c("normal")
denscomp(fln, legendtext = plot.legend)
qqcomp(fln, legendtext = plot.legend)

# Perform GofFit tests
gof_results <- gofstat(fln)
gof_results
gof_results$kstest

gof_results$chisq
gof_results$chisqpvalue
gof_results$chisqtable
```

```
     mean            sd
 -0.001518507    0.042617534
```

(b) **(8 Points)** Yes, for this data set the Normal distribution seems appropriate. As shown in Fig. 5, both the Q-Q plot and histogram suggest that the Normal distribution is a good fit for the log-returns. Note that having larger deviation around the tails on the Q-Q plot is expected since the variance is higher at extreme quantiles. Also, the KS test does not reject the Normal distribution at $(1 - \alpha) = 95\%$ confidence level.

Problem 4. (15 Pts.) Normal distribution is sometimes not a good fit for log-returns. Cauchy distribution is another (heavy-tailed) distribution used to model the log-returns. The CDF of the Cauchy distribution is given by:

$$F(x) = \frac{1}{\pi} \arctan\left(\frac{x - x_0}{\gamma}\right) + \frac{1}{2},$$

where x_0 and γ are parameters.

(a) Propose an inversion algorithm to generate samples of a Cauchy distribution with $x_0 = 0$ and $\gamma = 2$.

(b) Estimating the parameters of the Cauchy distribution using MLE is challenging. One approach to estimated the parameters using matching is to match median (1/2th quantile) and half of the inter-quartile range (difference between 1/4th and 3/4th quantiles) with their estimates from the data. Use this method to estimate the parameters using 1000 samples generated from part (a). How do they compare with the original parameters, i.e., $x_0 = 0$ and $\gamma = 2$?

(a) **(7 Points)** We denote the inverse cdf by $F^{-1}(.)$. So,

$$F(x) = u \iff x = F^{-1}(u).$$

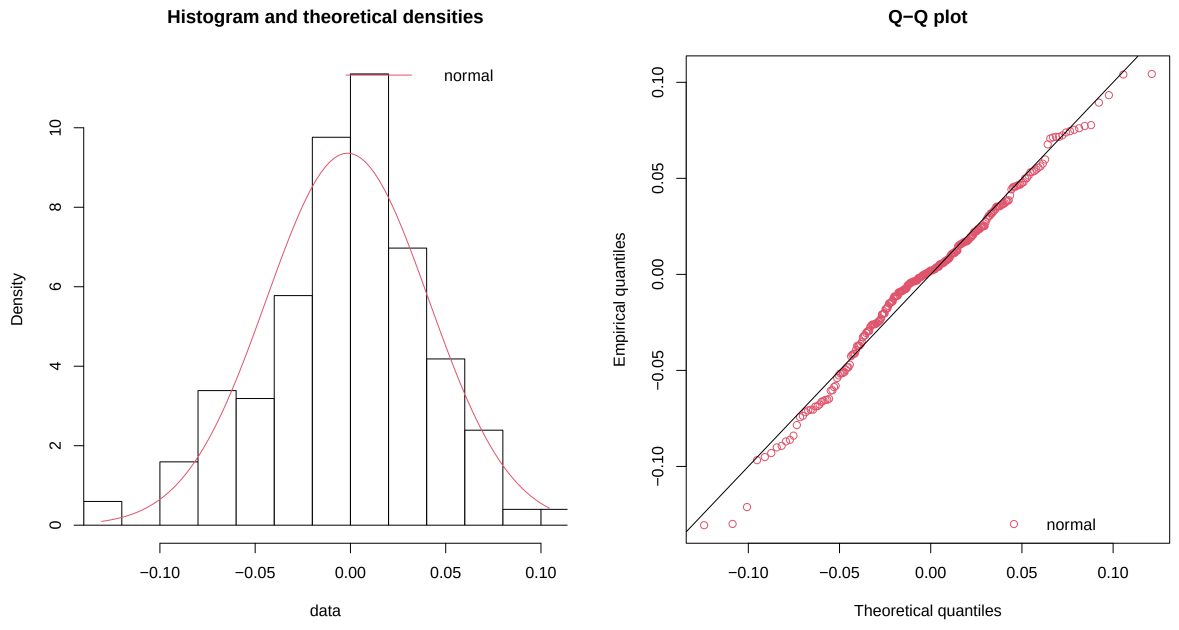

Figure 5: Graphical support for the goodness of fit of GBM.

We have

$$\frac{1}{\pi}\arctan\left(\frac{x}{2}\right)+\frac{1}{2}=u \implies \arctan\left(\frac{x}{2}\right)=\pi(u-\frac{1}{2}) \implies x=2\tan\left(\pi(u-\frac{1}{2})\right)$$

Therefore,

$$F^{-1}(u)=2\tan\left(\pi(u-\frac{1}{2})\right)$$

So, the inversion algorithm works as follows:

1. Generate $U \sim U[0,1]$.
2. Set $X=2\tan\left(\pi(U-\frac{1}{2})\right)$ and return X.

(b) **(8 Points)** We generate 1000 samples of the Cauchy distribution with $x_0=0$ and $\gamma=2$ and estimate median and half of the inter-quartile range to be 0.05 and 3.98, respectively.

For the Cauchy distribution, we can calculate the q-th quantile using $x_0+\gamma\tan\left(\pi(q-1/2)\right)$. So, matching median and half of the inter-quartile range with their estimates from the data, we have

$$x_0+\gamma\tan(0)=0.05 \implies x_0=0.05,$$

$$\gamma\tan\left(\frac{\pi}{4}\right)-\gamma\tan\left(-\frac{\pi}{4}\right)=3.98 \implies \gamma=1.99.$$

We observe that the estimated parameters are close enough to the original parameters.

```python
#  Cauchy distribution
import numpy as np

n = 1000
X_list = []
np.random.seed(1)
for i in range(n):
    U = np.random.random()
    X = 2*np.tan(np.pi*(U-1/2))
    X_list.append(X)

print("Estimate of median:", np.median(X_list))
print("Estimate of difference between 1/4th and 3/4th quantiles:", np.quantile
    (X_list, 3/4)-np.quantile(X_list, 1/4))
```

```
Estimate of median:  0.04713779707427136
Estimate of difference between 1/4th and 3/4th quantiles:  3.9842903109340155
```

Problem 5. **(20 Pts.)** Data on call counts to a call centre, by hour, for 1 month (31 days) are provided in CallCounts.xls. Let $N(t)$ represent the cumulative number of arrivals by time t. If the process is nonstationary Poisson, then $Var(N(t))/E(N(t)) = 1$ for all t, or stated differently $Var(N(t)) = \beta E(N(t))$ with $\beta = 1$. Since you have arrival count data, you can estimate $Var(N(t))$ and $E(N(t))$ at $t = 1, 2, \ldots, 8h$. Use these data to fit the regression model $Var(N(t_i)) = \beta E(N(t_i))$ and see if the estimated value of β supports the choice of a nonstationary Poisson arrival process. **Hints**: This is regression through the origin. Also, remember that $N(t_i)$ represents the total number of arrivals by time t_i.

(20 Points) We follow these steps to estimate β:

1. Transform hourly count data into cumulative count data by adding the number of calls in each hour to that of the previous hour.

2. For each hour $t_i, i \in \{1, 2, \ldots, 8\}$, estimate the expected cumulative arrivals up to that hour by computing sample average $\bar{N}(t_i)$ across the 31 days, i.e.,

$$\bar{N}(t_i) = \frac{1}{31}\sum_{j=1}^{31} N_j(t_i), \quad \forall i \in \{1, 2, \ldots, 8\},$$

   where $N_j(t_i)$ is the cumulative arrivals by hour t_i on day j.

3. Similarly, compute the sample variance $V(t_i)$ for each hour t_i, i.e.,

$$V(t_i) = \frac{1}{30}\sum_{j=1}^{31} (N_j(t_i) - \bar{N}(t_i))^2, \quad \forall i \in \{1, 2, \ldots, 8\}.$$

4. Perform regression of $V(t_i)$ on $\bar{N}(t_i)$ with zero intercept (8 data points) to estimate β.

```r
Data <- read.csv("CallCounts.csv", header=TRUE,
                 sep=",", na.strings=c("NA", ""), stringsAsFactors=FALSE, as.
                    is=TRUE)

# Create a new data frame with the same columns as Data
df <- data.frame(matrix(0, nrow=nrow(Data), ncol=ncol(Data)))

# Calculate cumulative sums for each column
for (i in 1:ncol(Data)) {
  df[,i] <- Data[,i]
  if (i > 1) {
    df[,i] <- df[,i] + df[,i-1]
  }
}

# Compute sample averages
sa <- apply(df, MARGIN = 2, mean)

# Compute sample variances
sv <- apply(df, MARGIN = 2, var)

# Perform linear regression and output 95% CI
df2 <- data.frame(sample_average = sa, sample_variance = sv)
df2.lm <- lm(sample_variance ~ sample_average - 1, data = df2) # zero
    intercept
summary(df2.lm)
confint(df2.lm, level = 0.95)
```

```
   2.5%      97.5%
 1.069191   1.368783
```

We find that the estimated coefficient is 1.22 with the 95% CI of $[1.07, 1.37]$. This is reasonably close to 1 which supports the use of nonstationary Poisson process. (You may also argue that based on the %95 CI the variability is larger than that of Poisson and hence a more variable arrival process would be more appropriate.)