Stochastic Simulation (MIE1613H) - Homework 3 (Solutions)

Due: March 21, 2022

Problem 1. (30 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:

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

Table 1: Number of available machines in each step.

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 $1/0.5512835 = 1.813949$.

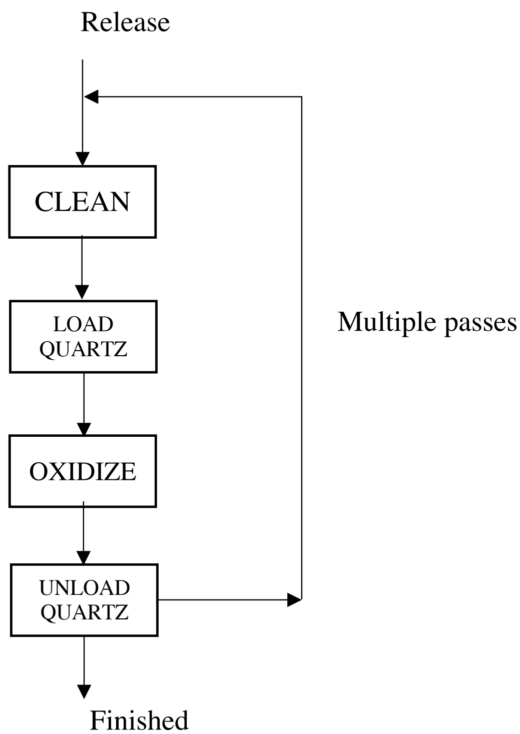

Figure 1: Diagram of the diffusion step.

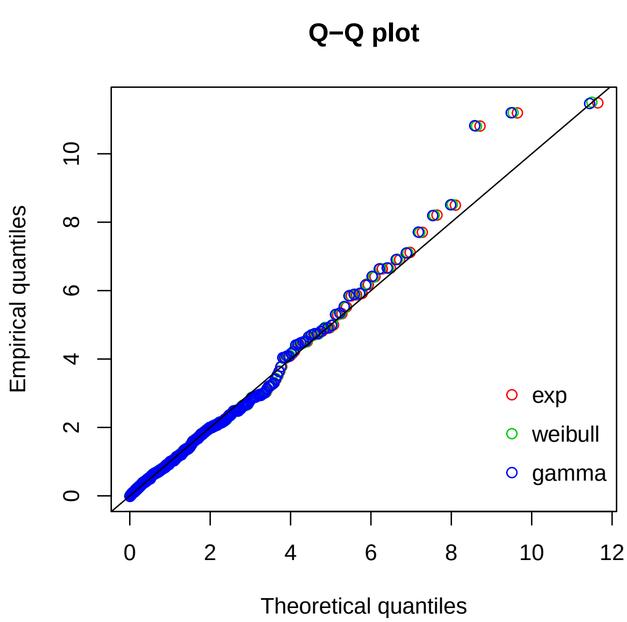

Figure 2: Q-Q plot for the fitted distributions to the CLEAN data.

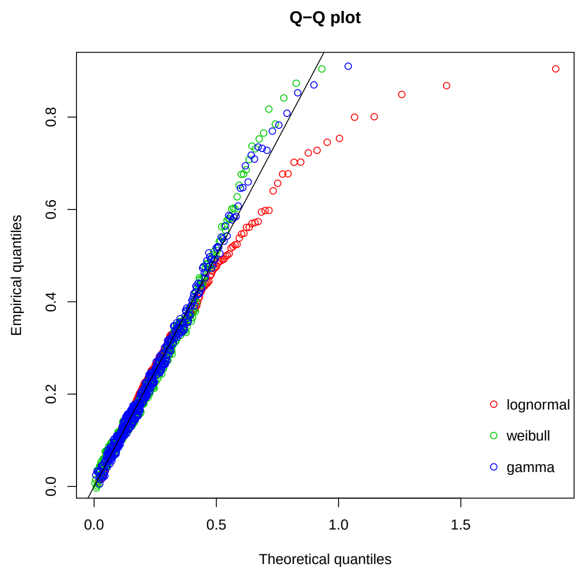

Figure 3: Q-Q plot for the fitted distributions to the LOAD data.

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 Lognromal 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")

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

```r
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) 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.

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.

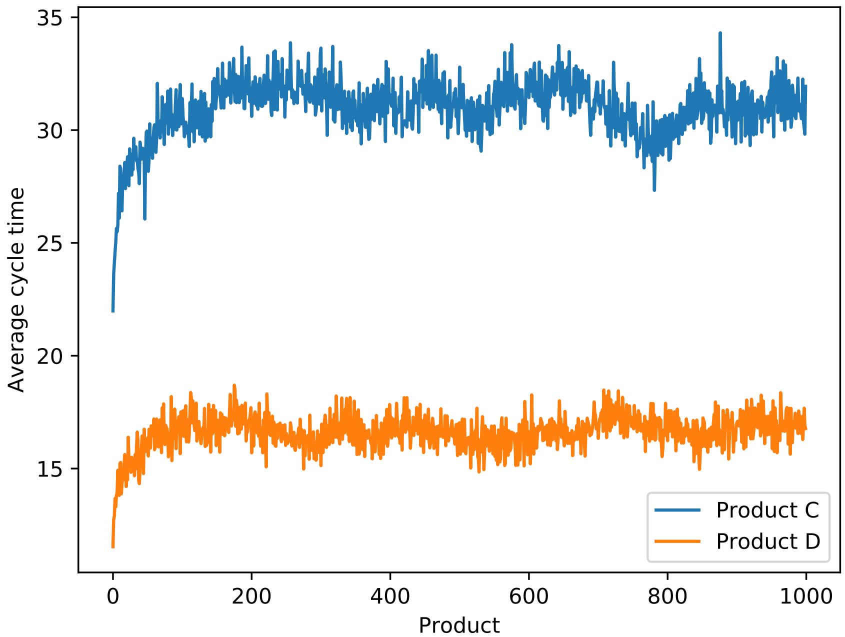

Figure 4: Mean plot, by product number, averaged across $n = 30$ replications.

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 [31.282, 31.309] and [16.693, 16.708], respectively.

```
# 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
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:
        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: (31.29586, 31.28239, 31.30934)
Estimated expected cycle time for product D: (16.70039, 16.69263, 16.70815)
```

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 98% 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.

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 0.98th quantile of the maximum number of calls reporting cardiac arrests during the day.

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

We simulate 1000 samples of the maximum number of calls during the day, sort them, and pick the 980^{th} value as the minimum number of drones required at the base for the desired service level. The estimated required drones is 224 with the 95% CI of $[Y_{(971)}, Y_{(989)}] = [222, 226]$.

```python
# 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 = []
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:
```

```python
        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 0.98th quantile
print("Estimated required drones is:", np.sort(AllMaxQueue)[980])
l = int(np.floor(980 - 1.96*np.sqrt(980*0.02)))
u = int(np.ceil(980 + 1.96*np.sqrt(980*0.02)))
print("The 95% CI: [{}, {}]".format(np.sort(AllMaxQueue)[l], np.sort(AllMaxQueue)[u]))
```

```
Estimated required drones is:  224
The 95% CI: [222, 226]
```

Problem 3. (20 Pts.) The attached file (TeslaPrices.csv) provides daily prices for Tesla's stock during 2021.

(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 $t_i, i = 1, \ldots, k$.

We first calculate the log-returns using the **Close** price data. We then fit a normal distribution to the log-returns data and estimate $\mu = 0.001525905$ and $\sigma = 0.034080501$.

```
# 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("TeslaPrices.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
0.001525905 0.034080501
```

(b) Does the GBM fit the data well? Use your estimates from part (a) together with both graphical methods and statistical goodness of fit tests discussed in class to justify your answer.

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.

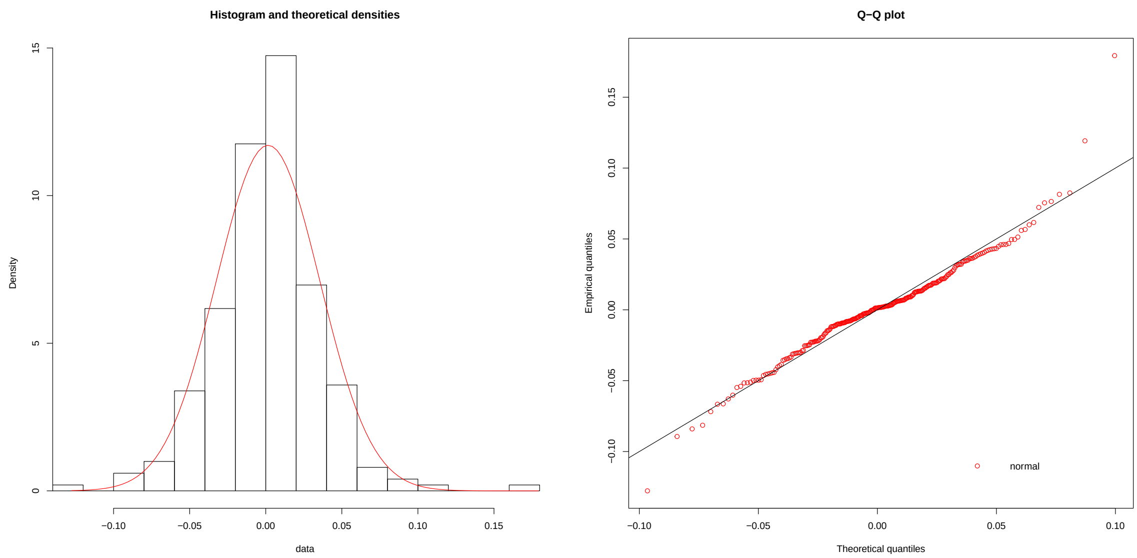

Figure 5: Graphical support for the goodness of fit of GBM.

Problem 4. (10 Pts.) Normal distribution is typically 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 $\gamma = 2$.

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

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

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) 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 $\gamma = 2$?

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 $x_0 + \gamma \tan(\pi(q - 1/2))$. 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.) The inversion method for generating a nonstationary arrival process requires an equilibrium base process with rate $\tilde{\lambda} = 1$ and variance σ_A^2. When $\sigma_A^2 > 1$, one approach is to use a balanced hyperexponential distribution. This means that $\tilde{A}$ is exponentially distributed with rate λ_1 with probability p, and exponentially distributed with rate λ_2 with probability $1 - p$. "balance" means that $p/\lambda_1 = (1 - p)/\lambda_2$. Thus, there are only two free parameters, p and λ_1. The following values of p and λ can be shown to achieve the desired arrival rate and variance (Exercise 34, Chapter 6):

$$p = \frac{1}{2}\left(1 + \sqrt{\frac{\sigma_A^2 - 1}{\sigma_A^2 + 1}}\right),\ \lambda_1 = 2p.$$

(a) Propose an inversion method to generate arrivals according to the cumulative arrival rate $\Lambda(t) = t^2$ and with a balanced hyperexponential base with $\sigma_A^2 = 1.4$.

HINT: Start by deriving the distribution of the first inter-arrival time distribution $G_e(t)$ for the equilibrium renewal process with balanced hyperexponential base. The CDF of the hyperexponential distribution with parameters p, λ_1, λ_2 is given by,

$$G(t) = p(1 - e^{-\lambda_1 t}) + (1 - p)(1 - e^{-\lambda_2 t}).$$

We have $1 - G(t) = pe^{-\lambda_1 t} + (1-p)e^{-\lambda_2 t}$, and therefore,

$$\begin{aligned}
G_e(t) &= \tilde{\lambda} \int_0^t (1 - G(s))ds \\
&= \int_0^t pe^{-\lambda_1 s} ds + \int_0^t (1-p)e^{-\lambda_2 s} ds \\
&= \frac{p}{\lambda_1}(1 - e^{-\lambda_1 t}) + \frac{1-p}{\lambda_2}(1 - e^{-\lambda_2 t}) \\
&= \frac{1}{2}(1 - e^{-\lambda_1 t}) + \frac{1}{2}(1 - e^{-\lambda_2 t}),
\end{aligned}$$

where we have used $\tilde{\lambda} = 1$ and $p/\lambda_1 = (1-p)/\lambda_2$. Note that the above implies that the initial inter-arrival time is equally likely to be exponential with rate λ_1 or λ_2. Therefore, it has a hyperexponential distribution with $p = (1-p) = 1/2$.

To generate a sample from a hyperexponential distribution, one can directly use the defenition. In the genertal case, consider a hyperexponential distribution with parameters p, λ_1, λ_2. That is, the random variable is exponentially distributed with rate λ_1, with probability p, and exponentially distributed with rate λ_2, with probability $(1-p)$. Then one can generate a sample X as follows:

1. Generate $U_1 \sim Unif[0,1]$ and $U_2 \sim Unif[0,1]$
2. If $U_1 \leq p$: Return $X = \frac{-1}{\lambda_1}\ln(U_2)$; else: Return $X = \frac{-1}{\lambda_2}\ln(U_2)$.

Note that the above returns an exponential with rate λ_1 with probability p, and an exponential with rate λ_2 with probability $(1-p)$. So, we generate the inter-arrival times $A_2, A_3, \ldots$ from the balanced hyper-exponential specified in the question and with parameters:

$$\begin{aligned}
p &= \frac{1}{2}\left(1 + \sqrt{\frac{1.4-1}{1.4+1}}\right) = 0.7, \\
\lambda_1 &= 2p = 1.4, \\
\lambda_2 &= (1-p)\lambda_1/p = 2(1-p) = 0.6.
\end{aligned}$$

The first inter-arrival time A_1 is generated from a hyper-exponential with

$$\begin{aligned}
p &= 0.5, \\
\lambda_1 &= 1.4, \\
\lambda_2 &= 0.6.
\end{aligned}$$

(b) Estimate $E(N(10))$ and $\text{Var}(N(10))$ using 100 samples generated using your algorithm in part (a) and compare the ratio $\text{Var}(N(10))/E(N(10))$ with the variance of the base equilibrium process. Explain your observation in a couple of sentences.

We generate 100 samples of $N(10)$ using the algorithm presented in part (a). The estimates of $E(N(10))$ and $\text{Var}(N(10))$ are 100.18 and 166.79, respectively. So, we estimate $\text{Var}(N(10))/E(N(10))$ to be 1.66 which is approximately close to the variance of the base equilibrium process.

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

```

```python
# the function returns a random variate
# from hyperexponential(q,l1,l2)

def HypExp(q,l1,l2):
    U1 = np.random.random()
    U2 = np.random.random()
    if U1 < q:
        return (-1/l1)*np.log(U2)
    else:
        return (-1/l2)*np.log(U2)

np.random.seed(1)

# set parameters of the renewal process
p = 0.7
lambda1 = 2*p
lambda2 = 2*(1-p)

N10_list = []
for i in range(100):
    # S_tilde denotes arrival times of the unit rate process
    # generate the first inter-arrival from Ge(t)
    S_tilde = HypExp(1/2, lambda1, lambda2)
    # S denotes the actual arrival times
    S = np.sqrt(S_tilde)
    # N denotes the arrival-counting process
    N = 0
    # generate arrivals during [0,10]
    while S <= 10:
        N += 1
        # generate the next arrival using G(t)
        S_tilde = S_tilde + HypExp(p,lambda1,lambda2)
        S = np.sqrt(S_tilde)

    N10_list.append(N)

print("Estimate of the Expectation:", np.mean(N10_list))
print("Estimate of the Variance:", np.var(N10_list))
print("Estimate of the Ration:", np.var(N10_list)/np.mean(N10_list))
```

```
Estimate of the Expectation:  100.18
Estimate of the Variance:  166.7876
Estimate of the Ration:  1.6648792174086642
```