COMP9334 report

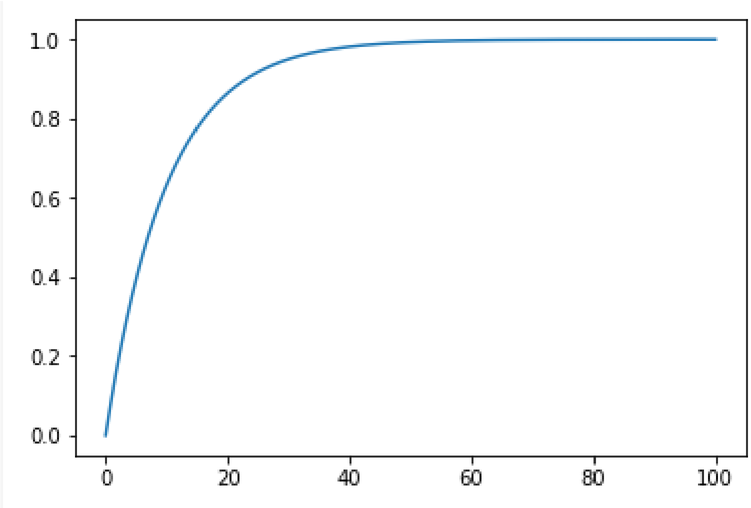
Generate inter-arrival probability distribution and service time distribution

Cumulative distribution function

If 6 jobs arrive every hour, it means that, on average one job arrives every 10 minutes. Let's define a variable $\lambda = \frac{1}{10}$ being the rate parameter. This rate parameter λ is a measure of frequency: the average rate of events (in this case, jobs) per unit of time (in this case, minutes).

Knowing this, the question that what the probability of a next job arrive within a curtain time is answered by a well-known function called **cumulative distribution function** of **exponential distribution**, and it looks like this:

$$F(x) = 1 - e^{-\lambda x}$$



As time passes, the probability of arrival increases towards one.

Generate random number from cumulative distribution function

A method to generate random number from a particular distribution is the *inverse transform method*:

- generate random floating point value **n** between **0** and **1** (*uniformly distribution*).
- compute the $F^{-1}(n)$

For **exponential distribution**, its **cumulative distribution function** *(CDF)* is

$$F(x) = 1 - e^{-\lambda x}$$

so the inverse of this *CDF* is

$$F^{-1}(x) = -log(1-x)/\lambda$$

x is uniformly distributed between $(0, 1)$

Generate exponential distributed random number using python

Here is one way to implement in python:

```
import math
import random

def nextTime(rateParameter):
    return -math.log(1.0 - random.random()) / rateParameter
```

Here is some sample output:

```
>>> nextTime(1/10.0)
6.579616062844679
>>> nextTime(1/10.0)
0.8496748345865709
>>> nextTime(1/10.0)
1.6038624255006428
>>> nextTime(1/10.0)
24.828638903932053
```

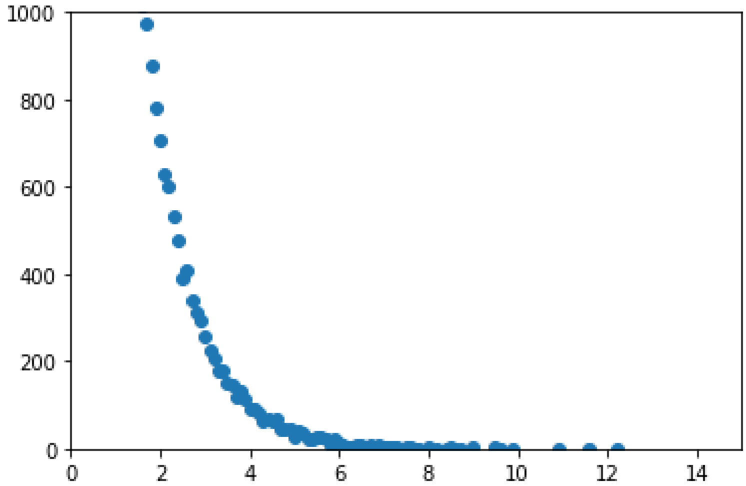
Let's run enough times to make sure that the average time arrival by this function really is 10

```
>>>sum([nextTime(1/10) for i in range(1000000)]) / 1000000
9.999013076811442
>>>sum([nextTime(1/10) for i in range(1000000)]) / 1000000
9.98838071976749
>>>sum([nextTime(1/10) for i in range(1000000)]) / 1000000
9.996894414471212
```

It very closes to what we want.

Another thing we can do is:

- Generate 50000 uniformly distributed numbers in $(0, 1)$,
- then compute $-log(1-x)/\lambda$,$\lambda = 10$,
- the plot shows below:



The histogram of the numbers generated in step 2 in many **bins**. The dots in graph show the expected number of *exponential distributed numbers* in each bin.

Now we can use this function to generate the *arrival time list* and *service time list* for the random mode test:

```
def generate_exp(T_end, arrival, service):
    arrival_list = []
    service_list = []
    arr , ser = 0.0, 0.0
    # next time = previous time + inter-arrival time generated
    arr += nextTime(arrival)
    while arr<= T_end:
        arrival_list.append(arr)
        # service time = sum fo three independent exponentially distributed numbers
        ser = nextTime(service)+nextTime(service)+nextTime(service)
        service_list.append(ser)
        arr += nextTime(arrival)
    return arrival_list,service_list
```

Simulation code verified

Using the example given in the project

- example 1:

m (*number of servers*) is 3 , T_c (*delayed off time*) is 100 and the *setup time* is 50. Here is the *arrival time* and *service time* table:

Arrival time	Service time
10	1
20	2
32	3
33	4

```
mrt is : 41.250
```

- example 2:

m (number of servers) is 3 , T_c (delayed off time) is 10 and the *setup time* is 5. Here is the *arrival time* and *service time* table:

Arrival time	Service time
11	1
11.2	1.4
11.3	5
13	1

```
mrt is : 2.275
```

Here is the two log files of simulation of these two examples:

> log1.txt log2.txt

Reproducibility

The *generate_exp()* function given before will generate different list each time because no fixed seed provided. In order to prove reproducibility, the function is changed a little bit as follow:

```
def generate_exp(T_end, arrival, service, fix = 0, seed = 0):
    arrival_list = []
    service_list = []
    arr , ser = 0.0, 0.0
    fix = int(fix)
    seed = int(seed)
    if fix == 1: # provide fixed seed so can generate same list each time
        random.seed(seed)
    arr += nextTime(arrival)
    while arr<= T_end:
        arrival_list.append(arr)
```

```
        ser = nextTime(service)+nextTime(service)+nextTime(service)
        service_list.append(ser)
        arr += nextTime(arrival)
    return arrival_list,service_list
```

Here is some sample output for a small test in *random_fix* mode, which is random mode with fixed seed so that *(arrive list, service list)* is the same:

```
>>> a_c = simulate('random_fix',0.35,1,5,5,0.1,100)
mrt is :6.178
>>> a,c = simulate('random_fix',0.35,1,5,5,0.1,100)
mrt is :6.178
>>> a,c = simulate('random_fix',0.35,1,5,5,0.1,100)
mrt is :6.178
```

and the *arrival list* and *complete list* (list contains the time each job completes) is:

```
>>>a_c = simulate('random_fix',0.35,1,5,5,0.1,100)
mrt is :6.178
>>>print(a_c[0])
[(5.31602031732921, 12.580004928056379), (7.361605316663926, 14.773651993798376),
(9.211329515387648, 16.545350137177063), (10.157214921039806, 16.872772968246686),
(19.915320911285267, 26.53481111463993), (18.089808550628085, 27.8415160617409‌7),
(17.029015485363168, 30.077375136240455), (27.754591813021236, 32.522036216576836),
(26.892390103029264, 33.89387752016058), (31.388624785005998, 35.876068715550026),
(31.391891850235623, 36.875806842326725), (32.515735831882246, 38.97015807567017),
(33.86419138414459, 39.696058449854995), (33.53426059528607, 40.18737605462384),
(33.294642561652694, 42.03599429971047), (38.67711075082986, 43.99135180993523),
(41.207976194899594, 44.01778468515373), (43.65711643059906, 44.75096772296489),
(46.36781545662845, 53.17824288655225), (50.416907469906725, 59.68893634723434),
(56.94399654391793, 63.52821450493707), (60.43468529209289, 67.41371241551136),
(66.87515990939391, 72.16264116103065), (73.09078713903698, 79.79191829937795),
(68.58619235358327, 81.31847161055705), (81.12387307843734, 82.77898958665679),
(76.288594988652, 83.16868100413538), (75.9326459104134, 83.25005237827565),
(81.25612755210626, 86.80410922647263), (84.51691903565222, 88.77129019842613),
(87.51776827040938, 93.41440995079586), (86.9680929164618, 93.6680237958128),
(88.23069710817259, 94.18891906380902), (87.07400933105534, 95.43949567834447),
(87.4519222450316, 96.6877706664429), (90.86722677020244, 97.24950347678353),
(90.23197358892396, 98.00360851372253)]
```

```
>>>a_c = simulate('random_fix',0.35,1,5,5,0.1,100)
mrt is :6.178
>>>print(a_c[0])
[(5.31602031732921, 12.580004928056379), (7.361605316663926, 14.773651993798376),
(9.211329515387648, 16.545350137177063), (10.157214921039806, 16.872772968246686),
(19.915320911285267, 26.53481111463993), (18.089808550628085, 27.84151606174097),
(17.029015485363168, 30.077375136240455), (27.754591813021236, 32.522036216576836),
(26.892390103029264, 33.89387752016058), (31.388624785005998, 35.876068715550026),
(31.391891850235623, 36.875806842326725), (32.515735831882246, 38.97015807567017),
(33.86419138414459, 39.696058449854995), (33.53426059528607, 40.18737605462384),
(33.294642561652694, 42.03599429971047), (38.67711075082986, 43.99135180993523),
(41.207976194899594, 44.01778468515373), (43.65711643059906, 44.75096772296489),
(46.36781545662845, 53.17824288655225), (50.416907469906725, 59.68893634723434),
(56.94399654391793, 63.52821450493707), (60.43468529209289, 67.41371241551136),
(66.87515990939391, 72.16264116103065), (73.09078713903698, 79.79191829937795),
(68.58619235358327, 81.31847161055705), (81.12387307843734, 82.77898958665679),
(76.288594988652, 83.16868100413538), (75.9326459104134, 83.25005237827565),
(81.25612755210626, 86.80410922647263), (84.51691903565222, 88.77129019842613),
(87.51776827040938, 93.41440995079586), (86.9680929164618, 93.6680237958128),
(88.23069710817259, 94.18891906380902), (87.07400933105534, 95.43949567834447),
(87.4519222450316, 96.6877706664429), (90.86722677020244, 97.24950347678353),
(90.23197358892396, 98.00360851372253)]
```

Given the same *arrival list*, it will output the same answer. The reproducibility is proven.

Here are three more reproducibility tests:

```
test 3:
mode,servers,setup_time,delayoff_time,time_end,arr_list,ser_list:
random_fix 3 50 100 1000 0.35 1.0
finished!
mrt is :5.908

test 4:
mode,servers,setup_time,delayoff_time,time_end,arr_list,ser_list:
random_fix 4 30 50 5000 0.7 1.0
finished!
mrt is :3.572

test 5:
mode,servers,setup_time,delayoff_time,time_end,arr_list,ser_list:
random_fix 5 5 10 1000 0.5 1.0
finished!
mrt is :3.922
```

Determining a suitable value of T_c

determining a suitable value of T_{end}(*length of simulation*):

choose the given value in the project document: the *number of servers* is 5, *setup time* is 5, $\lambda = 0.35$, $\mu = 1$, assume $T_c = 0.1$. T_{end} starts from 100 to 20100

T_{end}	**100**	**2100**	**4100**	**6100**	**8100**	**10100**	**12100**	**14100**	**16100**	**18100**	**20100**
res	5.480	6.072	6.026	5.931	6.064	6.110	6.049	6.072	6.036	6.112	6.129

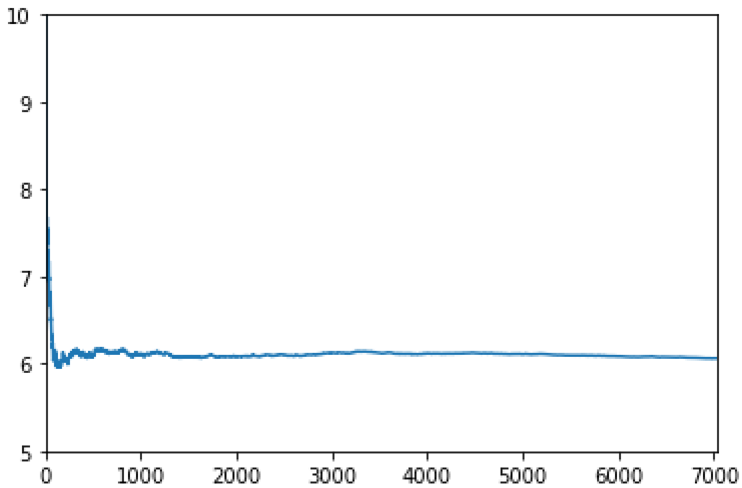
It shows that after $T_{end} = 8100$, the response time is around 6.06 stably. So we choose $T_{end} = 20000$ in case.

determining the *number of replications* (n):

$T_{end} = 20000$, considering the running time of the simulation, just starting from $n = 10$.

determining the *end of transient* (l):

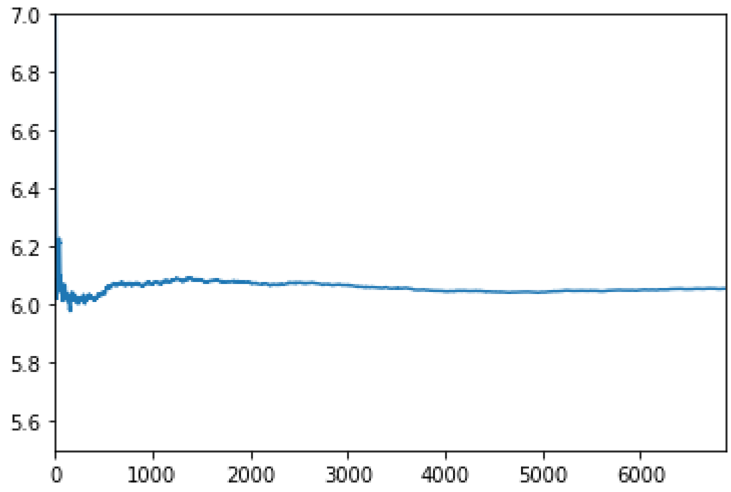
without removing the transient, one of simulation graph is this:



x-axis is the T_{end} and y-axis is mrt *(mean response time)*

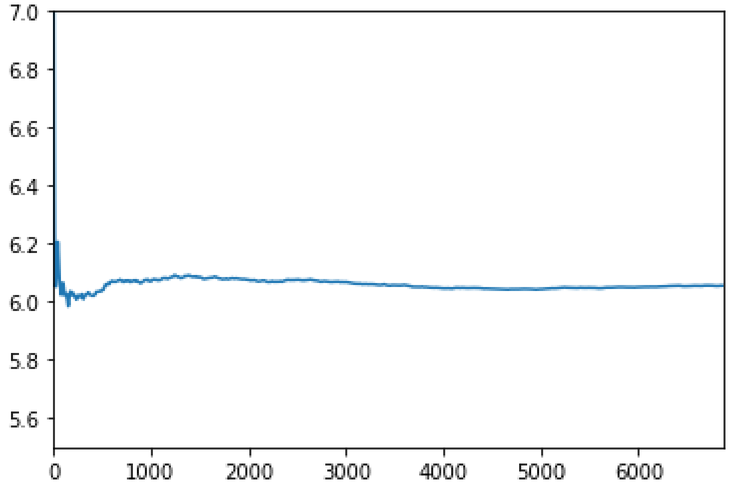
Using a program that implements the transient removal procedure by **Law** and **Kelton**. A parameter ω can be varied to get a smoothed curve.

- when $\omega = 1$, the curve shows:



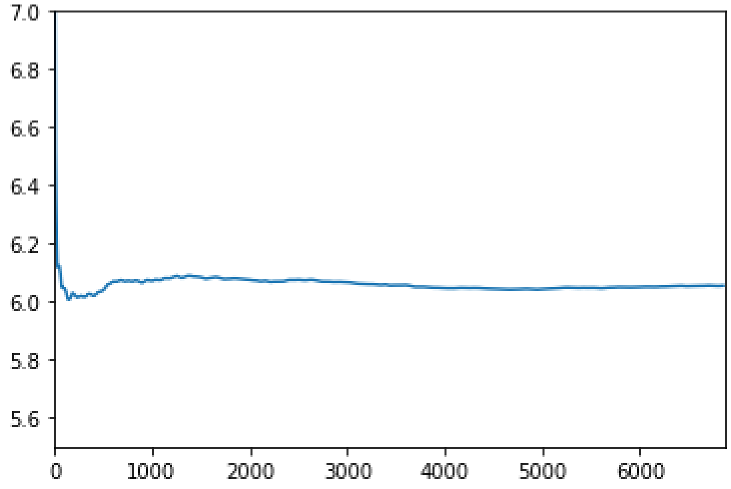
There are some oscillation in the graph so ω still need to be bigger.

- when $\omega = 5$, the curve shows:



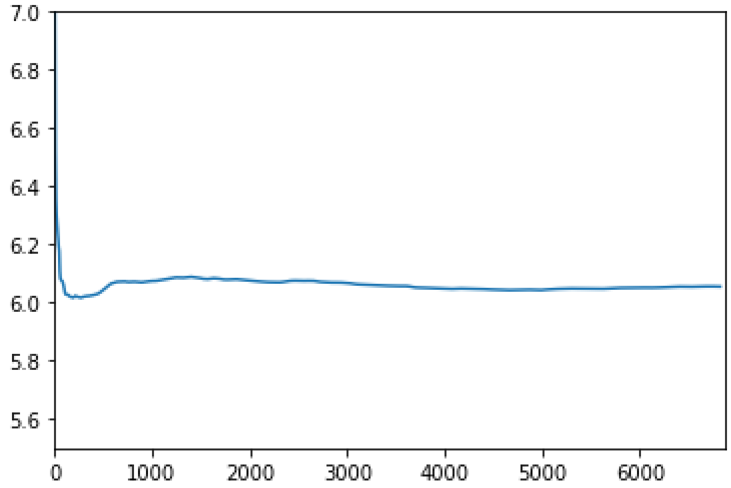
Still some oscillation in the graph so ω still need to be bigger.

- when $\omega = 20$, the curve shows:



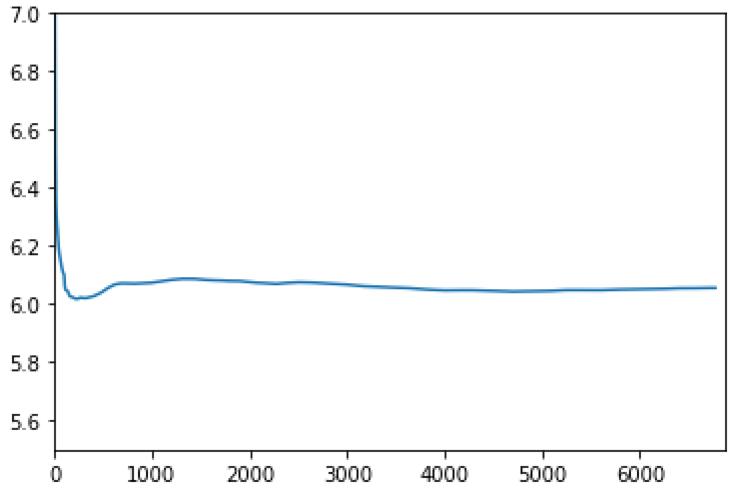
This is better but still not enough.

- when $\omega = 50$, the curve shows:



Much better but just some more test in case.

- when $\omega = 100$, the curve shows:



It's good enough. So the l (*end of transient*) $= 100$ After transient removal,

- the *sample mean* of $(n =)$ 10 replications $= 6.05788588625563$
- the *sample standard deviation* of 10 replications is 0.01587562344671423
- to compute the 95% confidence interval, $a = 0.05$
- Since having done 10 independent experiments and wanting 95% confidence interval, using $t_{9,0.975} = 2.262$
- the 95% *confidence interval* is: $[6.05788588625563 - 2.262\frac{0.01587562344671423}{\sqrt{10}}, 6.05788588625563 + 2.262\frac{0.01587562344671423}{\sqrt{10}}]$

95% *Confidence interval of mean response time*$=$ $[6.046529938392863, 6.069241834118398]$ is small enough.

So the parameters chosen are: *number of servers* is 5, *setup time* is 5, $\lambda = 0.35$, $\mu = 1$,$T_{end} = 20000$, *number of replications* (n) $= 10$.

determining a suitable value of T c

For the system with these parameters and $T_c = 0.1$, we infer that the reason of high response time is the low T_c causing the system need to be re-setup frequently. In order to get a improved system, which has lower mrt, the T_c need to be higher.

When we use a $T_c = 1$ system, summarise the data in a table:

- EMRT = estimated mean response time

\	**EMRT System 1($T_c = 0.1$)**	**EMRT System 2($T_c = 1$)**	**EMRT System 2 - EMRT System 1**
Rep. 1	6.103755337833227	5.667756688308885	-0.43599864952434153
Rep. 2	6.036903920786047	5.7375017806318835	-0.29940214015416355
Rep. 3	6.01941523556987	5.679789508618821	-0.33962572695104853
Rep. 4	6.059636558900138	5.683994529193845	-0.37564202970629257
Rep. 5	6.1308281500007	5.644545486272564	-0.48628266372813567
Rep. 6	6.062622154824946	5.705916908843001	-0.35670524598194486
Rep. 7	6.061100456436297	5.640056761050419	-0.4210436953858787
Rep. 8	6.0369751961009035	5.723134546912336	-0.3138406491885677
Rep. 9	5.979168710456368	5.690162346478765	-0.28900636397760326
Rep. 10	6.036846181790869	5.668442272176481	-0.36840390961438807

Compute the 95% confidence interval of for the last column (difference between 2 systems) is :

$$[-0.37119130260730415, -0.3659989122351688]$$

So we can say System 2 is better than System 1 with probability 95%. Our aim is to find a 2 units less than the system with $T_c = 0.1$, so still need to increase T_c.

When $T_c = 10$:

\	EMRT System 1($T_c = 0.1$)	EMRT System 3($T_c = 10$)	EMRT System 3 - EMRT System 1
Rep. 1	6.103755337833227	4.059404314942124	-2.044351022891103
Rep. 2	6.036903920786047	4.040642601926197	-1.9962613188598501
Rep. 3	6.01941523556987	4.040792035302791	-1.9786232002670792
Rep. 4	6.059636558900138	4.038982217265912	-2.020654341634226
Rep. 5	6.1308281500007	4.053719258290984	-2.077108891709716
Rep. 6	6.062622154824946	4.028697553290952	-2.033924601533994
Rep. 7	6.061100456436297	4.036543042435068	-2.0245574140012295
Rep. 8	6.0369751961009035	4.08661866826957	-1.9503565278313335
Rep. 9	5.979168710456368	4.0869327697588105	-1.8922359406975575
Rep. 10	6.036846181790869	4.033203153386797	-2.0036430284040723

Compute the 95% confidence interval of for the last column (difference between 2 systems) is :

$$[-2.003931854286826, -2.000411403279206]$$

So we can say System 3 is 2 units better than System 1 with probability 95%.

Still increase T_c up to 20 in case:

\	EMRT System 1($T_c = 0.1$)	EMRT System 4($T_c = 10$)	EMRT System 3 - EMRT System 1
Rep. 1	6.103755337833227	3.5659221195190454	-2.5378332183141814
Rep. 2	6.036903920786047	3.5404910115171333	-2.496412909268914
Rep. 3	6.01941523556987	3.517620800474077	-2.5017944350957926
Rep. 4	6.059636558900138	3.586713535421931	-2.472923023478207
Rep. 5	6.1308281500007	3.556310312317443	-2.574517837683257
Rep. 6	6.062622154824946	3.5629866548341043	-2.4996354999908417
Rep. 7	6.061100456436297	3.5756041785043338	-2.4854962779319636
Rep. 8	6.0369751961009035	3.529322491845899	-2.5076527042550043
Rep. 9	5.979168710456368	3.5794976080108256	-2.3996711024455424
Rep. 10	6.036846181790869	3.5413540282355234	-2.495492153555346

Compute the 95% confidence interval of for the last column (difference between 2 systems) is :

$$[-2.4984309802036546, -2.495854852200156]$$

System 4 is almost 2.5 units better than System 1 with probability 95%.

So we can say if our aim is to find a 2 units less than the system with $T_c = 0.1$, **when the system with $T_c > 10$, it is improved by 2 units with probability 95%**.