Laboratory Report Nº 1

Discrete Event Simulation

Group 6

Felix Saraiva 98752
Miguel Dias Gaio 82608
Miguel Ribeiro 87553

2021

Table of Contents

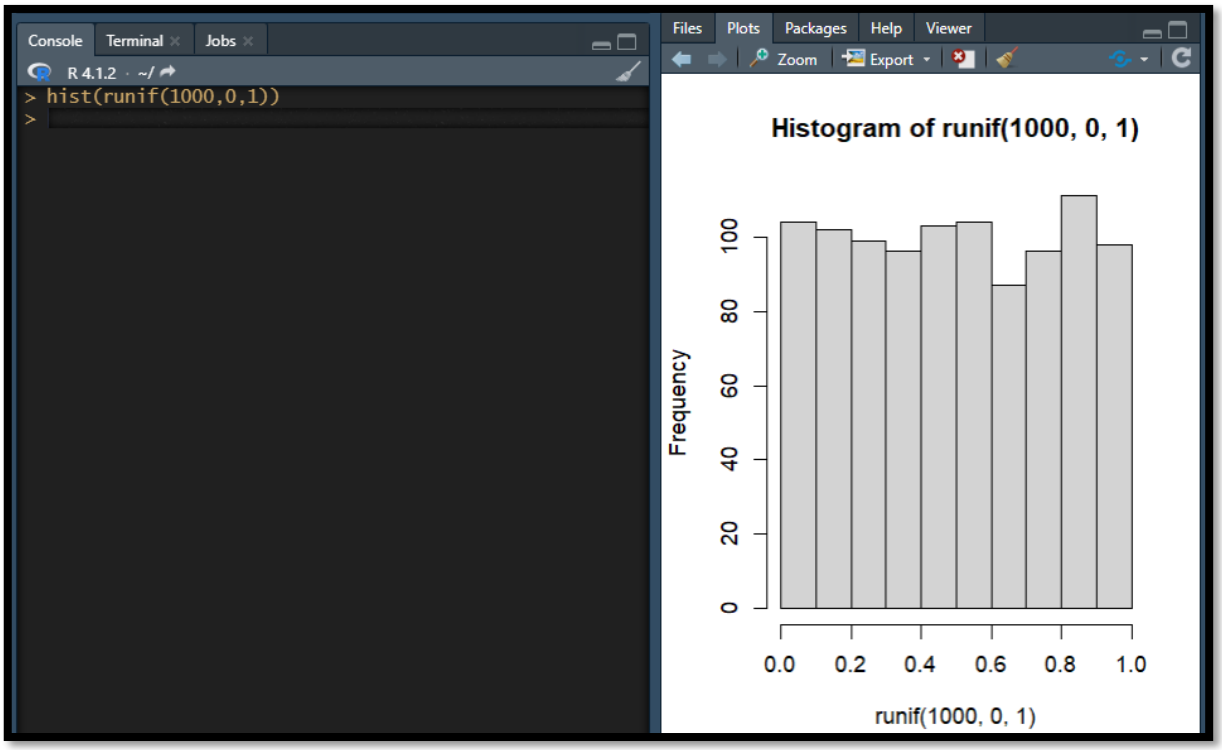
Objectives:

The aim of this laboratory work is to study the techniques for generating random variables and stochastic processes, and for programming discrete event simulators and analyse its output data.

Exercises:

A. Random number generators:

1. The aim of this first exercise is to plot a histogram of 1000 random numbers. The result is the expected since we obtain close probabilities of getting every shown uniformly distributed number.

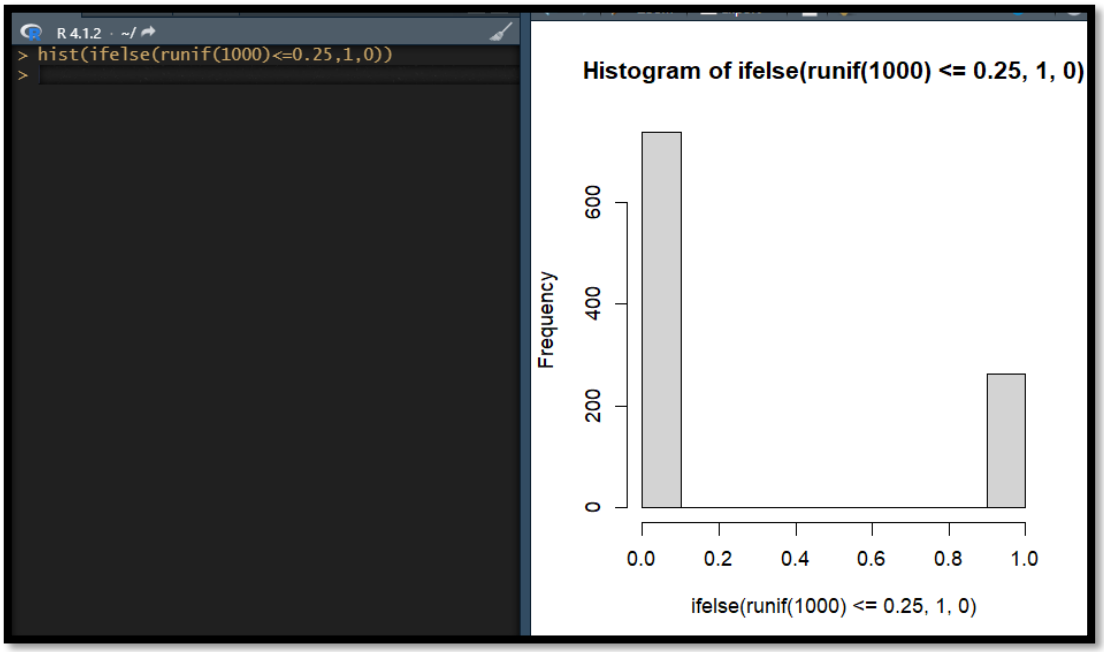


2. The aim of this exercise is to verify that the sequence of numbers generated by a pseudo-random number generator is in fact deterministic.
In the following example we prove that the random generator in R is not truly random by using the *set.seed()* function. This function places the generator in a specific point of its sequence.
By using the function, the second time, we put the generator in the same point it was before and as we can see we obtain the same random numbers, proving it is in fact deterministic.

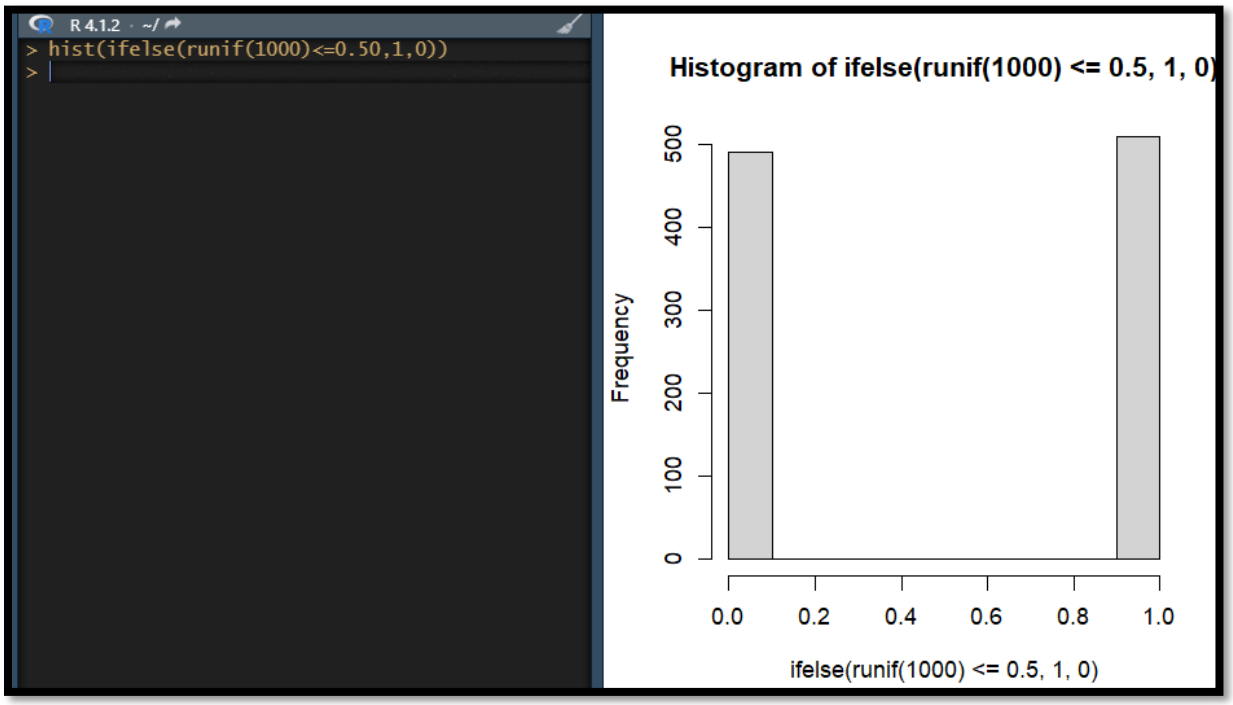
```
> set.seed(12)
> runif(5)
[1] 0.06936092 0.81777520 0.94262173 0.26938188 0.16934812
> runif(5)
[1] 0.033895622 0.178785004 0.641665366 0.022877743 0.008324827
> set.seed(12)
> runif(5)
[1] 0.06936092 0.81777520 0.94262173 0.26938188 0.16934812
> runif(5)
[1] 0.033895622 0.178785004 0.641665366 0.022877743 0.008324827
```

B. Generation of random variables and stochastic processes

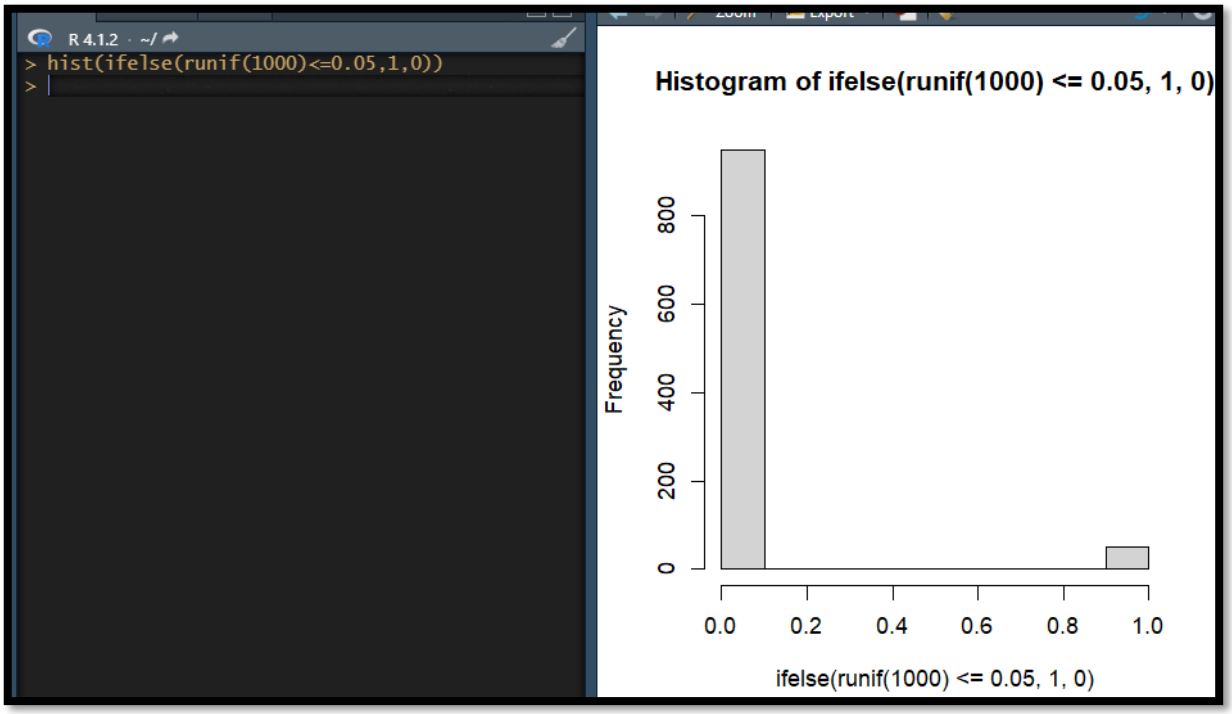
3. The goal of this exercise is to Develop an R script to generate Bernoulli distributed numbers using a random number generator and to plot a histogram of 1000 Bernoulli numbers to check (informally) that the generator is operating correctly.
We achieve this result by giving a condition to the probability of successes and by verifying that condition. In the examples bellow we always verify that the results on the histogram reflect the values of the success probability used.



25% Success Probability

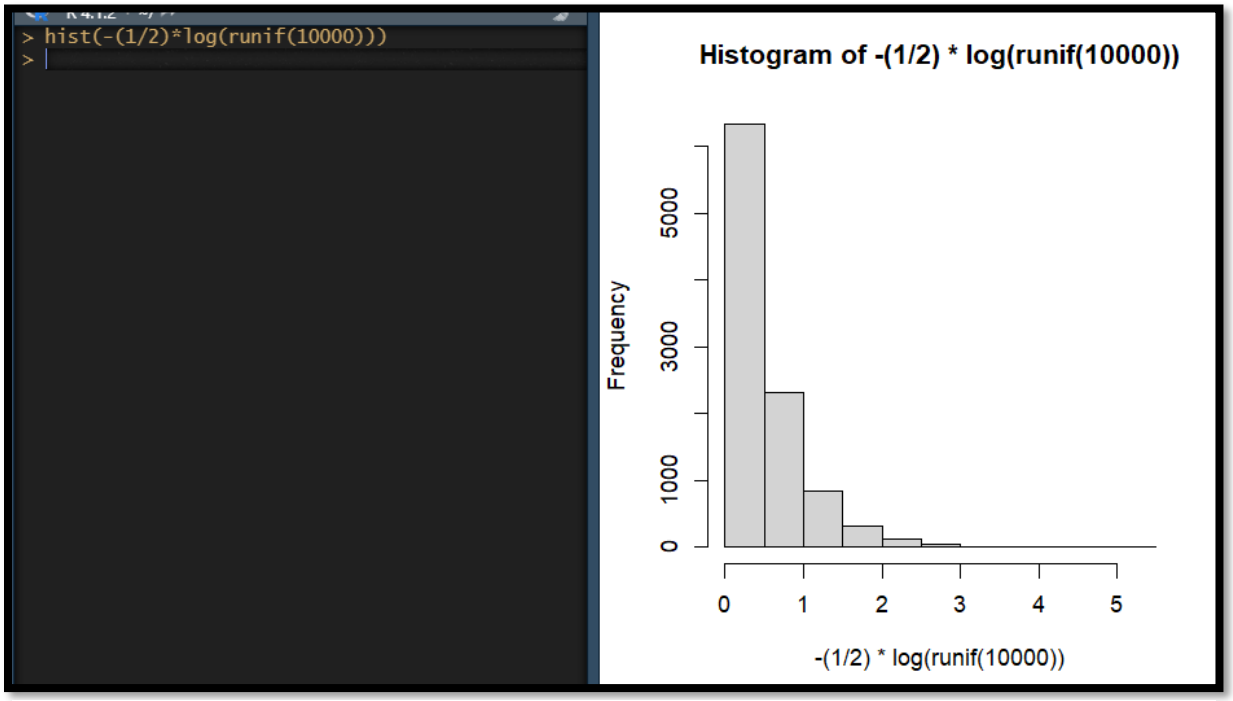


50% Success Probability

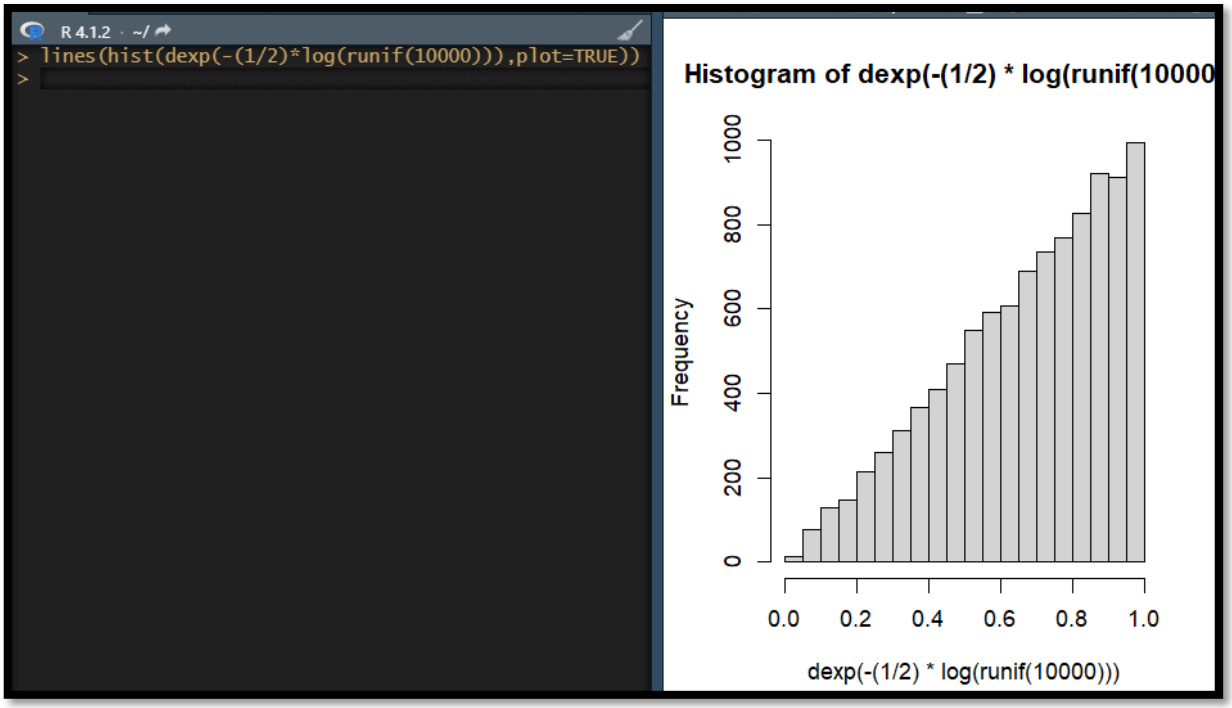


5% Success Probability

4. In this exercise we Developed an R script to generate exponentially distributed numbers using a random number generator and ploted a histogram of 10000 exponentially distributed numbers to check (informally) that the generator is operating correctly.
We can conclude that the generator is working correctly since we observe an exponential distributed groth in both graphs.

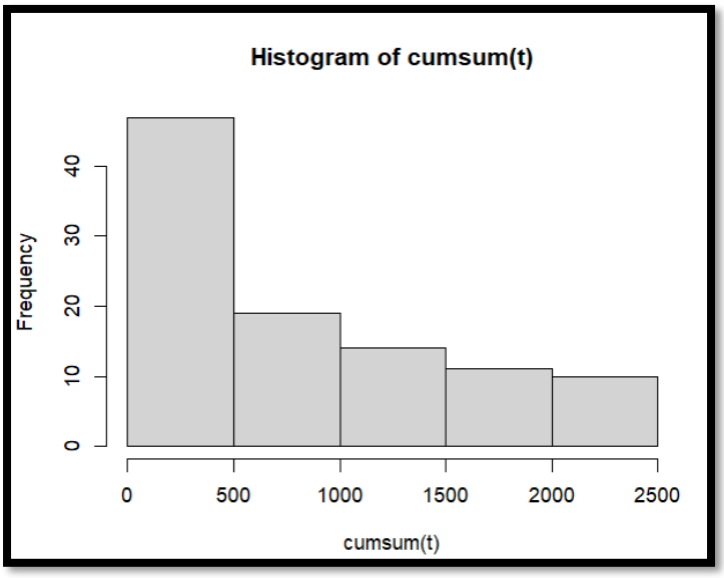


Histogram of 10000 exponentially distributed numbers



Probability Density Function values of the exponential distribution

5. The goal of this exercise is to Develop an R script to generate the arrival times of a Poisson process. The respective code of this exercise can be analysed on the *Annex A* of the report.



Poisson Exponentially distributed interarrival times graph (N=100, λ =2)

With these results we can conclude that a Poisson process has exponentially distributed interarrival times.

To verify that the sample rate is according to the population rate we have developed a test that consists in calculating the rate of data generated and compare it to the lambda value used in the experiment, if these two values match, then the experiment is correct.
The rate of the generated data is calculated by dividing the number of points generated (N) by the interval of time of the experiment, this interval of time is equal to the last generated arrival time.

- **Table of Test Results:**

Values	n= 100, λ=2	n= 100, λ=4	n= 1000, λ=6	n= 1000, λ=3
Expected	≈2	≈4	≈6	≈3
Practical	2.1	3.8	6.05	3.06

(We can also conclude that the higher the N the closer we get to the expected value)

C. Programming discrete event simulators

6. We are given a simulator responsible for estimating the average delay in queue of an **M/M/1** system. The goal of this exercise is to modify this simulator in order to estimate the **average delay in system** and its respective **server utilization**.

- **Average Delay in System Logic:**

  During our experiments the group found two solutions to this problem:

  1. In the first, and very similar to the already implemented average delay in queue algorithm, we would create a new AcumServerDelay variable and increment this variable with (Time - ServerArrivalTime), in the end the AvgServerDelay would be the AcumServerDelay/NumberofClients. The average delay in system would be calculated by both delays, queue and server.

  2. Although the first solution was more straightforward the group concluded that this second solution makes the code cleaner, less complex and with less variables.
  The implemented solution (in *Annex B*) presents a new simulator that does not focus on the queue and server separately but on the entire system as a whole.
  In this approach we create a *SystemArrivalTime* variable that stores the time when a client arrives at the system and use the `AcumSystemDelay=AcumSystemDelay+Time-SystemArrivalTime[1]` equation to accumulate the system delay when a client leaves the system. Then we reset the *SystemArrivalTime* and increment the NumSysCompleted variable that stores the number of clients that went through the system. The final average delay in system is calculated by:

  ```
  AvgSystemDelay=AcumSystemDelay/NumSysCompleted
  ```

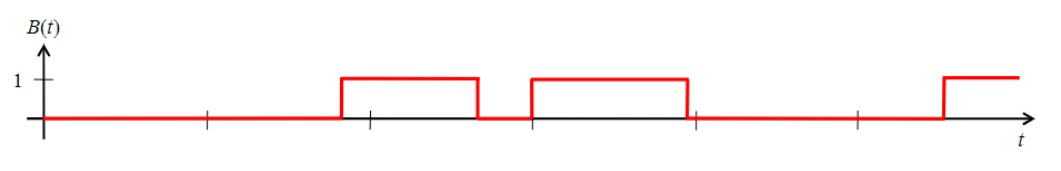
- **Server Utilization Logic:**

To calculate the Server Utilization, we calculated the time the server was busy (not idle) by applying the following:

```
AreaServerStatus=AreaServerStatus+ServerStatus*(Time-TimePreviousEvent)
```

and divided that value by the final *Time* of the system using:

```
ServerUtilization=AreaServerStatus/Time
```



Server Utilization Graph Example

Finally, we compared the values obtained with the theoretical ones, by trying different values on the attributes.

We calculated the theoretical value of the average system delay by using the following formula:

$$W = \frac{1}{\mu - \lambda}$$

Given the *ServiceRate* value and the *ArrivalRate* vale, we can estimate the theoretical value for the *AverageDelay*, by respecting the norms studied on lectures, which in this case is:

$$\frac{ArrivalRate}{ServiceRate} < 1$$

Values	λ= 1 μ= 2	λ= 4 μ=6	λ= 5 μ=5 (error)	λ=2 μ=1 (error)
Theoretical Value	1.0	0.5	1/0 —> infinite	-1
Practical Values	AvgDelay = 0.99 ServerUtilization = 0.49	AverageDelay = 0.47 ServerUtilization = 0.64	AverageDelay = 2.64 ServerUtilization = 0.90	AverageDelay = 248.38 ServerUtilization = 0.99

The respective code of this exercise can be verified on the *Annex B* of this report.

7. The goal of this exercise is to estimate the average delay in queue of an **M/M/2** system. The main challenge of this exercise was to introduce a new server to the system. This not only makes the system faster but also introduces the notion of overtaking where now clients can enter the queue and verify if there is any server available, if so then the client exits the queue and enters a server.
The respective code of the exercise can be verified on the *Annex C* of this report.
To check the theoretical values, we first calculated the utilization factor by doing:

$$\text{Utilization factor: } \rho = \lambda/m\mu$$

By trying different values of λ and μ such that $\lambda/2\mu < 1$ we have first chosen the values λ= 1 μ= 2

- Now, we use this recently discovered value to calculate the probability that an arrival finds all servers busy, using the following:

$$P_Q = \frac{(m\rho)^m}{m!(1-\rho)\left[\left(\sum_{i=0}^{m-1}\frac{(m\rho)^i}{i!}\right)+\frac{(m\rho)^m}{m!(1-\rho)}\right]}$$

- We calculate it by using *m* =2, since we have two servers, and then we use the respective result to calculate the:

$$\text{Average delay in queue: } W_Q = \frac{L_Q}{\lambda} = P_Q\frac{\rho}{\lambda(1-\rho)}$$

We present the theoretical and practical results in the following table:

Values	λ= 1 μ= 2	λ= 1/2 μ= 1
AvgDelay Theoretical Value	0.033	0.06
AvgDelay Practical Values	0.031	0.04

D. Estimation of probabilities

8. The following script:

```
1. N=1000
2. p=0.5
3. vecB=rbinom(N,1,p)
4. plot(cumsum(vecB)/c(1:N),type="l")
```

generates a sequence of N Bernoulli numbers with probability p and plots the relative frequency of successes as a function of N.
After repeating the plot several times, and for different values we conclude that the relative frequency is a good estimator of the success probability because the **Mean** of the relative frequency is equal to the probability p.
(Using larger *N* gets us closer to the true value of p.)

$$\mathrm{E}(Z(E)) = P(E)$$

And the **Variance** is equal to:

$$\mathrm{Var}(Z(E)) = \frac{P(E)\big(1 - P(E)\big)}{N}$$

Just Like we demonstrate in this R script:

```
> mean(cumsum(vecB)/c(1:N))
[1] 0.5170605
>
> var(cumsum(vecB)/c(1:N))
[1] 0.00261372
>
> (p*(1-p))/N
[1] 0.00025
>
```

9. The objective of this exercise is to develop an R script that simulates the tossing of five biased coins and estimates the probability of getting two heads when five coins are tossed, considering that the probability of getting a head is 0.7. (Code in Annex D)
Using the Relative Frequency (earlier prooved to be a good estimator) we calculated the probability of the experience and confirmed the correctness of the script by comparing the practical result with the exact values using the formula:

$$P(X) = \binom{n}{X} \cdot p^X \cdot (1-p)^{n-X}$$

Substituting in values for this problem, $n = 5$, $p = 0.7$, and $X = 2$.

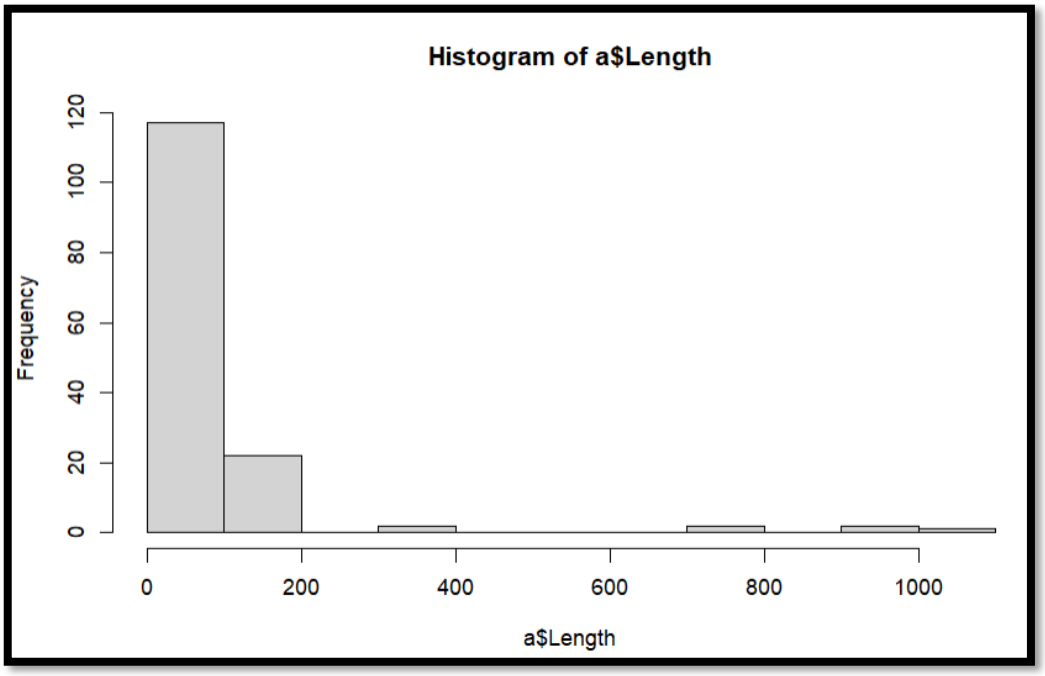
$$P(2) = \frac{5!}{2!(5-2)!} \cdot 0.7^2 \cdot (1-0.7)^{5-2}$$

	Probability
Exact Value	0.1323
Practical Value	0.1343

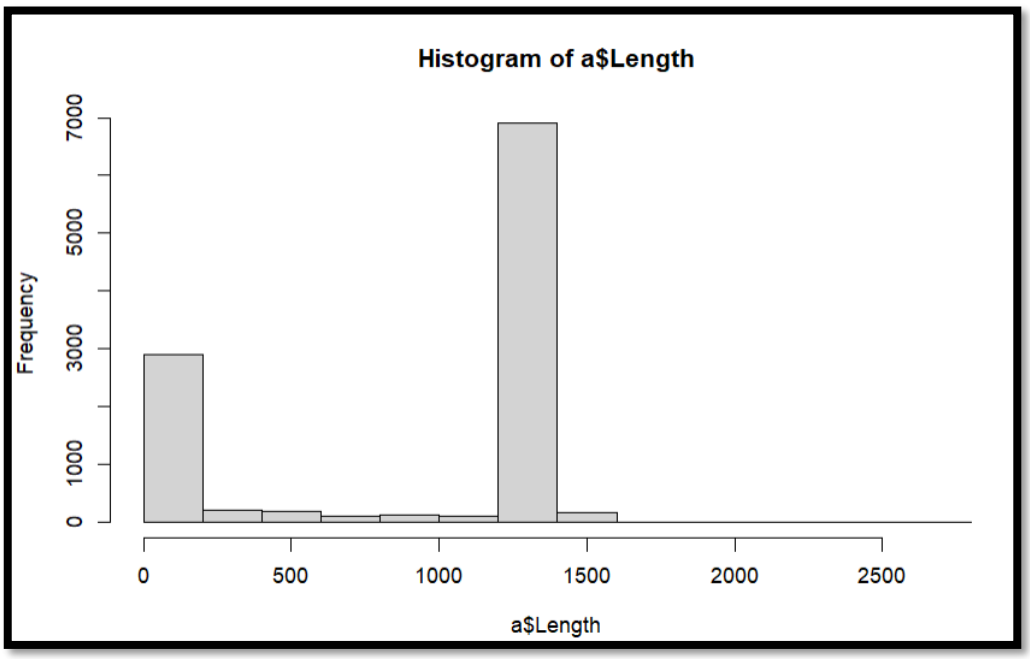
E. Revisions of statistics

10. In this exercise we analysed the distribution of packet sizes observed at a network interface, using *Wireshark* and *R*.
First, we captured traffic in the following situations:

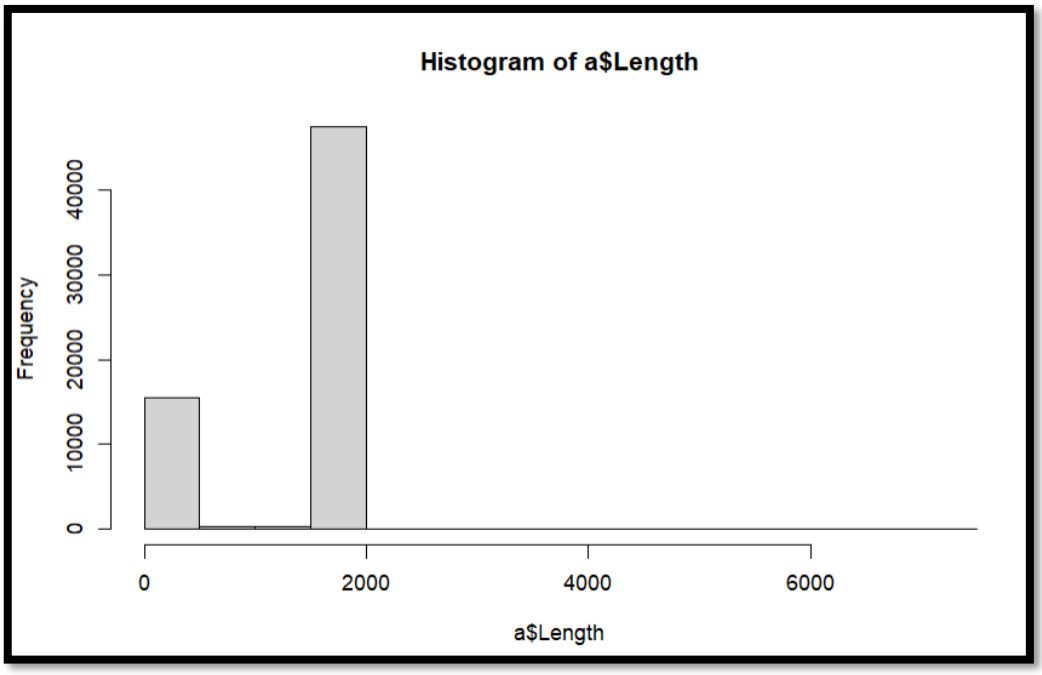
- While doing nothing:



- Watching a YouTube video:



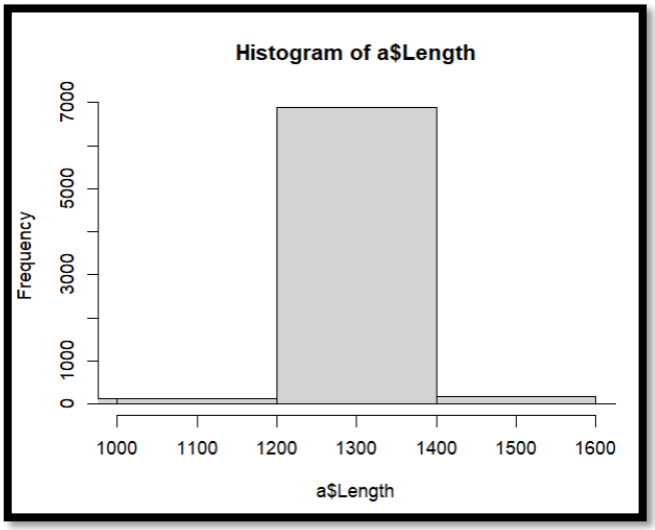
- Downloading a large file:



After capturing the packets and analysing the data of each situation, we have come to a conclusion regarding the different packet-sizes.

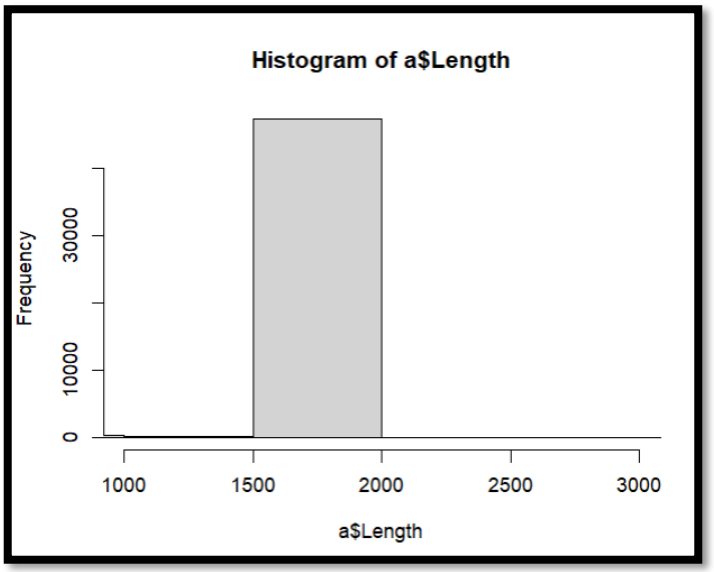
- While doing nothing the stream of data keeps a higher frequency of packets with sizes in the interval between [0-200] this happens because there is no particular application requesting or sending data.

- When we change to the situation of watching a YouTube video the frequency changes to a length in between sizes of [1200-1400].
  The packet-size distribution graph bellow shows a zoomed view of a sharp transition indicating that most of the packets are of a larger size than the previous situation.



Zoomed version of YouTube Video analysis

- Finally, when analysing the Download situation we can observe an even larger interval of packet-sizes, [1500-2000].
  As shown in the graph bellow an application requesting the download of a file uses even larger packets in fact, we can see a great predominance of packets with the maximum length.



Zoomed version of the Download analysis

- Code used in this exercise:

```
1. # Import the data
2. a <- read.csv(file = 'Download.csv')
3.
4. # Display Packet size
5. hist(a$Length,xlim = c(1000,3000))
6.
```

11. The goal of this exercise is to calculate two 95% confidence intervals for μ, one based on the Student's T distribution and the other based on the Normal distribution. Then we must indicate which interval we prefer. We are given a sample composed by 12 points.

- **Normal Distribution Formula:**

$$dNormal = Xn \pm z \times \sqrt{\frac{S^2(n)}{n}}$$

- **Student's T Distribution Formula:**

$$tStudent = Xn \pm t \times \sqrt{\frac{S^2(n)}{n}}$$

The code we used (located in *Annex E*) uses confidence intervals of *95%* to calculate both distributions.

The resulting confidence intervals when running the program were:

Normal Distribution	[0.66239088999929,2.35260911000071]
Student's t Distribution	[0.558465923638479,2.45653407636152]

By looking at the results we conclude that *Student's T Distribution* leads to a bigger interval, which in theory is better and represents the conclusion we were aiming for.

12. The goal of this exercise is to estimate the coverage of the 95% confidence interval for the mean based on 1000 experiments. We do this by using four different distributions: **Normal Distribution**, **Exponential Distribution** and **Lognormal Distribution** with variances of 1 and 2.

According to the **Central Limit Theorem**, the occurrence number should round the respective confidence interval (95%) as the number of samples increase.

Distributions	n= 5	n= 10	n= 50	n= 5000
Normal	0.956	0.933	0.951	0.948
Exponential	0.889	0.898	0.934	0.942
Lognormal $\sigma^2 = 1$	0.820	0.825	0.901	0.953
Lognormal $\sigma^2 = 2$	0.708	0.721	0.840	0.954

As it can be observed on the values presented above, the percentage rounds roughly 95% on all of them, it is even more accurate when *n* gets higher, as it was expected.

Also, we can conclude that the *Normal Distribution* is the only one closer to the needed percentage when it has a lower *n* value.

The respective code can be found on the *Annex F* of this report.

F. Output data analysis

13. In this exercise we made several runs on the R code of the simulator that estimates the average delay in queue of a M/M/1 queuing system, located in Annex G. The goal is to compare the values obtained in each simulation run and to conclude if one simulation run is enough to draw any conclusions.

Based on Queuing theory, we calculate the true queuing delay using:

$$W_Q = \frac{\lambda/\mu}{\mu - \lambda}$$

Table of Runs:

($\lambda = 1$ $\mu = 2$)

Nr. Of Run	1	2	3	4	5	6	7	8	9	10	True
Result	0.52	0.55	0.50	0.45	0.56	0.43	0.60	0.68	0.39	0.54	0.5

- After analysing the results, we can observe that although several values are very close to the true value, we can find some significant differences like the values 0.68 and 0.39.
  Regarding these results we conclude that one run is not enough, because that same run could differ greatly from the corresponding truth. This happens thanks to the use of random samples generated from probability distributions and these random variables may have large variances between each other.

14. In this exercise we Execute 25 runs of the M/M/1 simulation present in the Annex H (with λ = 3 and μ = 4), using two different run lengths (20 and 2000) and two different initial conditions (0 and 10 clients initially in the system). The goal is to evaluate the impact of the initial conditions in the performance estimates.

By analysing the obtained values present in the following table we can see that using a larger run length provides values that are closer to the true value, 0.75, and that the true value is indeed inside the confidence intervals. We can also conclude that when only considering the first 20 delays the estimate of the average queuing delay is very far from the true value, and 0.75 is not even inside the confidence intervals.

Run Length				
Initial Condition	**20**		**2000**	
(λ = 3 μ = 4)	**Mean**	**95% confidence interval**	**Mean**	**95% confidence interval**
0 Clients	0.343	[0.26-0.43]	0.72	[0.64-0.79]
10 Clients	1.86	[1.55-2.18]	0.77	[0.68-0.87]

- After this analyse, we can conclude that the initial delay values are biased in the result and in the situations where the system is initially empty these values are lower than the true value.

- In the higher run length situation by not using the initial values to calculate the queuing delay of the system, we are doing a simulation **warm-up,** and this gets us way closer to the true value.

15. In this exercise we analysed the number of required replicas based on the replication/deletion procedure applied to the estimation of the average queuing delay of an M/M/1 system.
We have done this analyse as a function of:

- ***(i)*** relative error
- ***(ii)*** warm-up period (number of delays deleted initially)
- ***(iii)*** number of delays used to compute the actual estimate
- ***(iv)*** and the ratio λ/μ.

Using the following example:

- ***(i)*** Relative error = 0.05
- ***(ii)*** Warm-up Period = 100
- ***(iii)*** Number of delays = 2000
- ***(iv)*** Ratio λ/μ = 1/2

With these values we require **20 Replicas**.

- **Relative error:**
  - When analysing the relative error, the group concluded that the smaller the relative error the larger number of replicas are required to stop the simulation.
  - Changing the Relative error to 0.01 we need 610 replicas.

- **Warm-up Period:**
  - The Warm-up Period showed us that the smaller the warm-up period the larger the number of replicas we need.
  - Changing the Warm-up Period = 50 we require 30 replicas.

- **Number of Delays:**
  - Studying the Number of delays the conclusion is that the smaller the number of delays used to compute the estimate the larger is the number of replicas used to stop the simulation.
  - Changing the Number of delays = 500 we require 113 replicas

- **Ratio:**
  - Relatively to the Ratio the larger the Arrival Rate the larger is the number of replicas required to stop the simulation.
  - Using a Ratio of 8/7, we need to use 225 replicas

G. Extra (optional) exercises

16. In this exercise we analyse the following example.
Urn 1 contains four blue balls and two yellow ones, while in Urn 2 the mix is three blue and five yellow. A ball is drawn at random from Urn 1, and then put into Urn 2. A ball is then drawn at random from Urn 2. We have developed the following R script to estimate the probability that the second ball drawn is blue and compared the result with the exact value.

```
1. n = 0
2. success = 0
3. #Runs the experience 1000 times
4. while (n<1000) {
5.   Urn1 <- c('B','B','B','B','Y','Y')
6.   Urn2 <- c('B','B','B','Y','Y','Y','Y','Y')
7.
8.
9.    Ball1 <- sample(Urn1, 1)
10.  Urn2 <- append(Urn2, Ball1)
11.  Ball2 <- sample(Urn2, 1)
12.
13.   if (Ball2 == 'B') {
14.     success= success+1
15.   }
16.   n=n+1
17. }
18. #Calculates the Probability
19. Probability = success/n
20.
21.
22. print(Urn2)
23. print(Probability)
24.
```

The exact value:

(Calculated using the law of total probability)

P = 0.7 x 0.4 + 0.3 x 0.3 = 0.28 + 0.09 = **0.37**

Value Obtained: 0.4

17. The objective of this exercise is to analyse the quality of the relative frequency as an estimator of the success probability of the Bernoulli random variable. To do so we have performed a large number of sets of experiments (500) where, in each set, N Bernoulli numbers are generated, and the relative frequency of successes is determined using the formula:

$$\frac{\text{number of times the event occured}}{\text{number of experiments}}$$

- We once more conclude that the Relative frequency is a good estimator of the success probability by verifying that the mean of the relative frequency of the event is equal to the probability of the event itself (0.5).

$$\mathrm{E}(Z(E)) = P(E)$$

- And that the variance is equal to:

$$\mathrm{Var}(Z(E)) = \frac{P(E)\big(1 - P(E)\big)}{N}$$

Table of Values:

	Mean	***Variance***
Exact Value	0.5	0.0125
Practical Value	0.49	0.0123

Code of the exercise present in Annex K.

References

- DDRS course notes, chapter 1 and 3, Rui Valadas
- "Simulation, Modeling and Analysis", Averill M. Law, McGraw-Hill, 4th edition, 2007
- DDRS videos and Presentations from Module 1, Rui Valadas
- Laboratory guide nº 1 – Discrete event simulation, 2021, Rui Valadas

Annex A

```
1. N = 1000
2. y = 3
3. t <- vector(length = N)
4. #Using the Poisson Formula
5. t[0] <- (-1 / y) * log(rexp(1))
6.
7. for (i in 1:N) {
8.   t[i + 1] <- t[i] + rexp(1,y)
9. }
10.
11. hist(cumsum(t))
12. SRatevsPRate = N / t[N]
13. print(SRatevsPRate)
14.
```

Annex B

```
1. ArrivalRate = 1
2. ServiceRate = 2
3. Time = 0
4. ServerStatus = 0
5. NumInQueue = 0
6. AcumSystemDelay = 0
7. TimePreviousEvent = 0
8. AreaServerStatus = 0
9. NumSysCompleted = 0
10. SystemArrivalTime = c()
11. EventList = c(rexp(1, ArrivalRate), Inf)
12. while (NumSysCompleted < 1000) {
13.   NextEventType = which.min(EventList)
14.   Time = EventList[NextEventType]
15.   AreaServerStatus = AreaServerStatus + ServerStatus * (Time
   - TimePreviousEvent)
16.   if (NextEventType == 1) {
17.     SystemArrivalTime = c(SystemArrivalTime, Time)
18.     EventList[1] = Time + rexp(1, ArrivalRate)
19.     if (ServerStatus == 1) {
20.       NumInQueue = NumInQueue + 1
21.     } else {
22.       ServerStatus = 1
23.       EventList[2] = Time + rexp(1, ServiceRate)
24.     }
25.   } else {
26.     AcumSystemDelay = AcumSystemDelay + Time -
   SystemArrivalTime[1]
27.     SystemArrivalTime = SystemArrivalTime[-1]
28.     NumSysCompleted = NumSysCompleted + 1
29.     if (NumInQueue == 0) {
30.       ServerStatus = 0
31.       EventList[2] = Inf
32.     } else {
33.       NumInQueue = NumInQueue - 1
34.       EventList[2] = Time + rexp(1, ServiceRate)
35.     }
36.   }
37.   TimePreviousEvent = Time
38. }
39.
40.
41. ServerUtilization = AreaServerStatus / Time
42. AvgSystemDelay = AcumSystemDelay / NumSysCompleted
43.
44. print(AvgSystemDelay)
45. print(ServerUtilization)
```

Annex C

```
ArrivalRate = 1
ServiceRate = 2
Time = 0
NumQueueCompleted = 0
ServerStatus1 = 0
ServerStatus2 = 0
NumInQueue = 0
AcumDelay = 0
QueueArrivalTime = c()
EventList = c(rexp(1, ArrivalRate), Inf,Inf)
while (NumQueueCompleted < 1000) {
  NextEventType = which.min(EventList)
  Time = EventList[NextEventType]
  if (NextEventType == 1) {
    #Event is arrival
    EventList[1] = Time + rexp(1, ArrivalRate) #Next arrival
    if (ServerStatus1 == 1) {
      #Server1 Busy
      if (ServerStatus2 == 1) {
        #Server2 Busy
        QueueArrivalTime = c(QueueArrivalTime, Time)
        NumInQueue = NumInQueue + 1
      }
      else{
        #Server2 idle
        NumQueueCompleted = NumQueueCompleted + 1
        ServerStatus2 = 1
        EventList[3]=Time+rexp(1,ServiceRate)#Next departure from 2
      }

    } else {
      #Server1 idle
      NumQueueCompleted = NumQueueCompleted + 1
      ServerStatus1 = 1
      EventList[2] = Time + rexp(1, ServiceRate)#Next departure from 1
    }
  } else {
    #Event is Departure
    if (NumInQueue == 0) {
      #Queue empty
      ServerStatus1 = 0
      ServerStatus2 = 0
      EventList[2] = Inf
      EventList[3] = Inf
    } else {
      #Queue not empty
      AcumDelay = AcumDelay + Time - QueueArrivalTime[1]
      QueueArrivalTime = QueueArrivalTime[-1]
      NumInQueue = NumInQueue - 1
      NumQueueCompleted = NumQueueCompleted + 1
      EventList[2] = Time + rexp(1, ServiceRate)#Next departure
    }
  }
}
AvgDelay = AcumDelay / NumQueueCompleted
AvgQueue=mean(AvgDelay)
cat("Average Delay in Queue: ",AvgQueue)


TheoryValueinQueue=(2*(ArrivalRate/(2*ServiceRate))^3)/(ArrivalRate*(1-
  (ArrivalRate/(2*ServiceRate)))*(1+(ArrivalRate/(2*ServiceRate))))
cat("Theory value of Delay in Queue: ",TheoryValueinQueue)
```

Annex D

```
1. a <- c()
2. SuccessMean = 0.4
3. N = 0
4. NrofSuccessTrials = 0
5.
6. while (N < 1000) {
7.   a <- rbinom(5, 1, 0.7)
8.   m <- mean(a)
9.   print(a)
10.   if (m==SuccessMean) {
11.     NrofSuccessTrials=NrofSuccessTrials+1
12.   }
13.   N = N + 1
14. }
15.
16. RelativeFrequency=NrofSuccessTrials/1000
17.
18. print(RelativeFrequency)
19.
20.
```

Annex E

```
1. v=c(1.58,-1.04,2.11,0.41,3.40,-0.34,-0.12,
   3.14,3.12,1.54,1.53,2.76)
2.
3. m<-mean(v)
4. s<-var(v)
5.
6. norm<-qnorm(0.975)
7. confN<-norm*sqrt(s/12)
8. st<-qt(0.975,df = length(v)-1)
9. confT<-st*sqrt(s/12)
10.
11. print(paste0("[",m-confN,",",m+confN,"]"))
12. print(paste0("[",m-confT,",",m+confT,"]"))
13.
```

Annex F

```
#Estimates the coverage of confidence intervals

E=1000 #Number of experiments
SizesOfSamples=c(5,10,50,5000) #Sizes of samples
tcritical=c(qt(0.975,df=SizesOfSamples[1]-1),
qt(0.975,df=SizesOfSamples[2]-1),
qt(0.975,df=SizesOfSamples[3]-1),
qt(0.975,df=SizesOfSamples[4]-1)) #Critical points t
distribution

#Standard normal distribution
mu=0
InInt=rep(0,4)
for (i in 1:4) {
  N=SizesOfSamples[i]
  for (j in 1:E) {
    a=rnorm(N)
    ma=mean(a)
    va=var(a)
    alinf=ma-tcritical[i]*sqrt(va/N)
    alsup=ma+tcritical[i]*sqrt(va/N)
    if (mu>alinf & mu<alsup) InInt[i]=InInt[i]+1
  }
}
CICoverage=InInt/E
print("Standard Normal distribution")
print(CICoverage)

#Exponential distribution
mu=1
InInt=rep(0,4)
for (i in 1:4) {
  N=SizesOfSamples[i]
  for (j in 1:E) {
    a=rexp(N)
    ma=mean(a)
    va=var(a)
    alinf=ma-tcritical[i]*sqrt(va/N)
    alsup=ma+tcritical[i]*sqrt(va/N)
    if (mu>alinf & mu<alsup) InInt[i]=InInt[i]+1
  }
}
CICoverage=InInt/E
print("Exponential distribution")
print(CICoverage)


```

```r
#Standard Lognormal distribution with sigma^2=1
mu=exp(1/2)
InInt=rep(0,4)
for (i in 1:4) {
  N=SizesOfSamples[i]
  for (j in 1:E) {
    a=rlnorm(N)
    ma=mean(a)
    va=var(a)
    alinf=ma-tcritical[i]*sqrt(va/N)
    alsup=ma+tcritical[i]*sqrt(va/N)
    if (mu>alinf & mu<alsup) InInt[i]=InInt[i]+1
  }
}
CICoverage=InInt/E
print("Standard Lognormal distribution sigma^2=1")
print(CICoverage)

#Standard Lognormal distribution with sigma^2=2
mu=exp(1)
InInt=rep(0,4)
for (i in 1:4) {
  N=SizesOfSamples[i]
  for (j in 1:E) {
    a=rlnorm(N,meanlog=0,sdlog=sqrt(2))
    ma=mean(a)
    va=var(a)
    alinf=ma-tcritical[i]*sqrt(va/N)
    alsup=ma+tcritical[i]*sqrt(va/N)
    if (mu>alinf & mu<alsup) InInt[i]=InInt[i]+1
  }
}
CICoverage=InInt/E
print("Standard Lognormal distribution sigma^2=2")
print(CICoverage)


```

Annex G

- Provided by the Professor
  R code of the simulator that estimates the average delay in queue of a M/M/1 queuing system

```
1. ArrivalRate = 1
2. ServiceRate = 2
3. Time = 0
4. NumQueueCompleted = 0
5. ServerStatus = 0
6. NumInQueue = 0
7. AcumDelay = 0
8. QueueArrivalTime = c()
9. EventList = c(rexp(1, ArrivalRate), Inf)
10. while (NumQueueCompleted < 1000) {
11.   NextEventType = which.min(EventList)
12.   Time = EventList[NextEventType]
13.   if (NextEventType == 1) {
14.     EventList[1] = Time + rexp(1, ArrivalRate)
15.     if (ServerStatus == 1) {
16.       QueueArrivalTime = c(QueueArrivalTime, Time)
17.       NumInQueue = NumInQueue + 1
18.     } else {
19.       NumQueueCompleted = NumQueueCompleted + 1
20.       ServerStatus = 1
21.       EventList[2] = Time + rexp(1, ServiceRate)
22.     }
23.   } else {
24.     if (NumInQueue == 0) {
25.       ServerStatus = 0
26.       EventList[2] = Inf
27.     } else {
28.       AcumDelay = AcumDelay + Time - QueueArrivalTime[1]
29.       QueueArrivalTime = QueueArrivalTime[-1]
30.       NumInQueue = NumInQueue - 1
31.       NumQueueCompleted = NumQueueCompleted + 1
32.       EventList[2] = Time + rexp(1, ServiceRate)
33.     }
34.   }
35. }
36. AvgDelay = AcumDelay / NumQueueCompleted
37.
```

Annex H

```
1. fmm1_initcond = function(ArrivalRate,
2.                          ServiceRate,
3.                          NumEndClients,
4.                          InitClients) {
5.   #Estimates the average delay in queue of an M/M/1 queuing system,
   with the
6.   #possibility of defining initial conditions
7.   #NumEndClients: stopping condition; number of clients that
   traversed the queue
8.   #InitClients: number of clients initially in the system
9.   NumQueueCompleted = 0
10.    ServerStatus = ifelse(InitClients == 0, 0, 1)
11.    NumInQueue = ifelse(InitClients == 0, 0, InitClients - 1)
12.    AcumDelay = 0
13.    Time = 0
14.    QueueArrivalTime = c()
15.    if (InitClients == 0) {
16.      EventList = c(rexp(1, Time + ArrivalRate), Inf)
17.    } else {
18.      for (i in 1:(InitClients - 1)) {
19.        Time = Time + rexp(1, ArrivalRate)
20.        QueueArrivalTime = c(QueueArrivalTime, Time)
21.      }
22.      EventList = c(Time + rexp(1, ArrivalRate), Time + rexp(1,
   ServiceRate))
23.    }
24.    while (NumQueueCompleted < NumEndClients) {
25.      NextEventType = which.min(EventList)
26.      Time = EventList[NextEventType]
27.      if (NextEventType == 1) {
28.        EventList[1] = Time + rexp(1, ArrivalRate)
29.        if (ServerStatus == 1) {
30.          QueueArrivalTime = c(QueueArrivalTime, Time)
31.          NumInQueue = NumInQueue + 1
32.        } else {
33.          NumQueueCompleted = NumQueueCompleted + 1
34.          ServerStatus = 1
35.          EventList[2] = Time + rexp(1, ServiceRate)
36.        }
37.      } else {
38.        if (NumInQueue == 0) {
39.          ServerStatus = 0
40.          EventList[2] = Inf
41.        } else {
42.          AcumDelay = AcumDelay + Time - QueueArrivalTime[1]
43.          QueueArrivalTime = QueueArrivalTime[-1]
44.          NumInQueue = NumInQueue - 1
45.          NumQueueCompleted = NumQueueCompleted + 1
46.          EventList[2] = Time + rexp(1, ServiceRate)
47.        }
48.      }
```

```
49.    }
50.    AcumDelay / NumQueueCompleted
51.  }
52.
53.  #Executes the simulation 25 runs
54.  N = 25
55.  a <- vector(length = N)
56.  for (i in 1:N) {
57.    a[i + 1] <- fmm1_initcond(3, 4, 2000,10)
58.
59.  }
60.  #Calculates mean and Variance of the average delay in queue
61.  mean = mean(a)
62.  variance = var(a)
63.
64.  #calculates percentage critical point
65.  percent <- 0.95
66.  p <- 1 - ((1 - percent) / 2)
67.
68.  #Calculates lower and Upper bound of Confidence Interval using
   Student's T Distribution
69.  intervalT <- qt(p, 24) * sqrt(variance / 25)
70.  higherpoint <- mean + intervalT
71.  lowerpoint <- mean - intervalT
72.
73.  cat("Points in Experience: \n", a)
74.  cat("\nLower Bound in Confidence interval: ", lowerpoint)
75.  cat("\nUpper Bound in Confidence interval: ", higherpoint)
76.  cat("\nMean = ", mean)
77.
```

Annex J

```
1. frepdel =
   function(gamma=0.05,ArrivalRate=1,ServiceRate=2,NumDeletions=100,
2.                     NumDelays=2000,conf=0.95,n0=10) {
3.   #Replication/deletion procedure for stopping simulations; returns
   number of
4.   #required replicas
5.   #gamma: relative error
6.   #n0: number of initial replicas
7.   #conf: confidence of the confidence interval
8.   #NumDeletions: number of delays deleted initially, in each replica
9.   #NumDelays: stopping condition for each replica; number of delays
   used to
10.    #compute the actual estimate
11.
12.    delays=c()
13.    for (i in 1:n0) {
14.
   delays=c(delays,fmm1_wu(ArrivalRate,ServiceRate,NumDeletions,NumDela
   ys))
15.    }
16.    md=mean(delays)
17.    tcritical=qt(1-(1-conf)/2,df=n0-1)
18.    vd=var(delays)
19.    l12=tcritical*sqrt(vd/n0)
20.
21.    n=n0
22.    while (l12/md > gamma/(gamma+1)) {
23.      n=n+1
24.
   delays=c(delays,fmm1_wu(ArrivalRate,ServiceRate,NumDeletions,NumDela
   ys))
25.      md=mean(delays)
26.      tcritical=qt(1-(1-conf)/2,df=n-1)
27.      vd=var(delays)
28.      l12=tcritical*sqrt(vd/n)
29.    }
30.
31.    infCI=md-l12
32.    supCI=md+l12
33.
34.    #cat(sprintf("Number of replicas = %d",n),"\n")
35.    #cat(sprintf("Mean delay = %f",md),"\n")
36.    #cat(sprintf("Half-length of confidence interval = %f",l12),"\n")
37.    #cat(sprintf("Confidence interval is [%f,%f]",infCI,supCI),"\n")
38.
39.    return(n)
40.  }
41.
42.
```

```
43.  fmm1_wu = function(ArrivalRate,ServiceRate,NumDeletions,NumDelays)
   {
44.    NumQueueCompleted=0
45.    ServerStatus=0
46.    NumInQueue=0
47.    AcumDelay=0
48.    QueueArrivalTime=c()
49.    EventList=c(rexp(1,ArrivalRate),Inf)
50.    while (NumQueueCompleted<(NumDeletions+NumDelays)) {
51.      NextEventType=which.min(EventList)
52.      Time=EventList[NextEventType]
53.      if (NextEventType==1) {
54.        EventList[1]=Time+rexp(1,ArrivalRate)
55.        if (ServerStatus==1) {
56.          QueueArrivalTime=c(QueueArrivalTime,Time)
57.          NumInQueue=NumInQueue+1
58.        } else {
59.          NumQueueCompleted=NumQueueCompleted+1
60.          ServerStatus=1
61.          EventList[2]=Time+rexp(1,ServiceRate)
62.        }
63.      } else {
64.        if (NumInQueue==0) {
65.          ServerStatus=0
66.          EventList[2]=Inf
67.        } else {
68.          NumQueueCompleted=NumQueueCompleted+1
69.          if (NumQueueCompleted>NumDeletions)
   AcumDelay=AcumDelay+Time-QueueArrivalTime[1]
70.          QueueArrivalTime=QueueArrivalTime[-1]
71.          NumInQueue=NumInQueue-1
72.          EventList[2]=Time+rexp(1,ServiceRate)
73.        }
74.      }
75.    }
76.    return(AcumDelay/NumDelays)
77.  }
78.
79.  frepdel()
80.
```

Annex K

```
1. N=20
2. Experiments=0
3. Probability=0.5
4. BernoulliNumbers = c()
5. RelativeFrequencyCollection= c()
6.
7. while (Experiments<500) {
8.
9.   BernoulliNumbers <- rbinom(N,1,Probability)
10.
11.   #Calculates nr of Ones in a set
12.   NrOfSuccess=length(BernoulliNumbers[BernoulliNumbers > 0])
13.   #Calculates Relative Frequency of set and stores it for
   final calculations
14.   RelativeFrequency = NrOfSuccess/N
15.   RelativeFrequencyCollection<-
   append(RelativeFrequencyCollection, RelativeFrequency)
16.
17.   Experiments=Experiments+1
18. }
19.
20. mean<-mean(RelativeFrequencyCollection)
21. variance<-var(RelativeFrequencyCollection)
22.
23. cat("\nMean of Relative Frequency of Successes: ",mean)
24. cat("\nVariance of Relative Frequency of Successes:
   ",variance)
25.
```