Stochastic Processes: Assignment 1

Group 1: Javier Esteban Aragoneses, Mauricio Marcos Fajgenbaun, Danyu Zhang, Daniel Alonso

January 10th, 2020

Importing libraries

```
#> Package:  markovchain
#> Version:  0.8.5-2
#> Date:     2020-09-07
#> BugReport: https://github.com/spedygiorgio/markovchain/issues
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

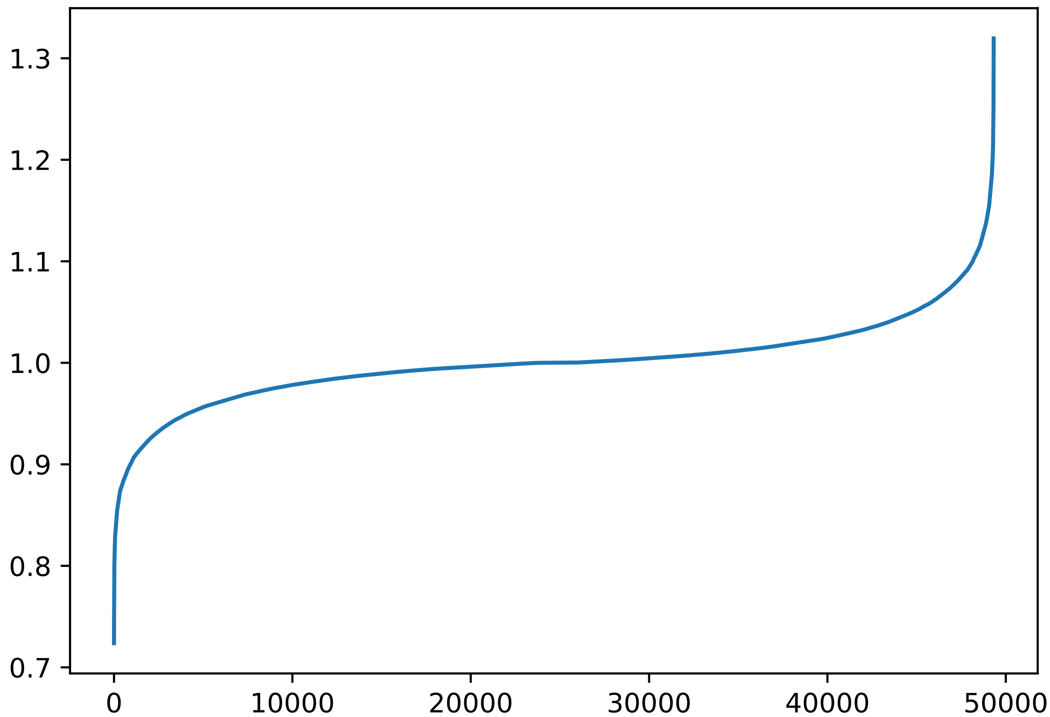
Problem 1

```
# importing libraries
import numpy as np
from copy import deepcopy
import pandas as pd
import matplotlib.pyplot as plt
from nltk.corpus import words

# Importing the matrix with the frequencies
freq =  np.loadtxt('./data/Englishcharacters.txt', usecols=range(27))

# transformation function for table values
def transf_log(table):
    return np.log(table + 1)

# applying the transformation function
freq = transf_log(freq)

# importing the messages and only selecting the
# message with index 0, as this is the one corresponding
# to group 1
with open('./data/messages.txt') as f:
    message = f.readlines()[0].replace('\n','')

# decode function
def decode(message, freqs, iters):
```

```
    """
    Function to decode a message enconded
    using a substitution cipher utilizing the
    Metropolis-Hastings algorithm.

    Params:
    message = string to decode
    freqs = frequency table for transitions of letters
            to input
    iters = amount of iterations to perform
    """
    # dictionary to organize the iterations
    # the score and the result of the attempt
    # to decode the message corresponding to that
    # iteration
    msg_iters = {}

    # defining the identity function
    # all letters to be used excluding spaces
    letters = ["a", "b", "c", "d",
               "e", "f", "g", "h",
               "i", "j", "k", "l",
               "m", "n", "o", "p",
               "q", "r", "s", "t",
               "u", "v", "w", "x",
               "y", "z"]
    # creating a copy of the original letters
    # to use as key for the dictionaries
    init_letters = deepcopy(letters)
    # creating a dictionary with init_letters as keys
    # and letters as values
    cd = {l:d for l,d in zip(init_letters,letters)}
    # every time we update with {' ':' '} we add the space
    # to the dictionary
    cd.update({' ':' '})
    # define a function that just uses the previous
    # dictionary to seek the letters
    def f(c):
        return cd[c]

    # this dictionary and subsequent function maps each letter
    # to a column/row in the freq matrix, ex: 'a':0, 'b':1
    fvals = np.array([x for x in range(len(letters))])
    cd_map = {l:v for l,v in zip(letters,fvals)}
    cd_map.update({' ':26})
    def f_map(c):
        return cd_map[c]

    # score function uses sum of logs
    def score(fun):
        p = 0
        for i in range(1,len(msg)):
            p = p + freq[f_map(fun(msg[i-1])),f_map(fun(msg[i]))]
```

```
        return p

    # converting the message to a list in order to
    # go through the letters in pairs
    msg = list(message)

    # letters list, this one shall be modified
    # every time the score passes the test
    letters_n = deepcopy(letters)

    # loop iters amount of times
    for i in range(iters):
        # randomly choose 2 numbers and replace the 2 chosen
        # vals in a copy of letters
        ch1 = np.random.randint(0,len(letters))
        ch2 = np.random.randint(0,len(letters))
        plc1 = deepcopy(letters_n[ch1])
        plc2 = deepcopy(letters_n[ch2])
        letters_n[ch1] = plc2
        letters_n[ch2] = plc1

        # create the dictionary for the f* function
        cd_n = {l:v for l,v in zip(init_letters,letters_n)}
        # add the space to it after scramble
        cd_n.update({' ':' '})
        # f* definition
        def f_n(c):
            return cd_n[c]

        # calculating the score for each function and its ratio
        scr_f = score(f)
        scr_fn = score(f_n)
        a = scr_fn/scr_f
        # test if a random number is lower than min(a, 1)
        cond = np.random.rand() <= min(a,1)
        # if condition is true
        if cond:
            # replacing the letters list with the one from f*
            letters = deepcopy(letters_n)
            # updating the dictionary with the new letters list after replacing
            cd = {l:v for l,v in zip(init_letters,letters)}
            cd.update({' ':' '})
            # f re-definition
            def f(c):
                return cd[c]
            # replacing the letters in the message using
            # the new f replaced by f*
            for k in range(len(msg)):
                msg[k] = f(msg[k])
            # adding score and joining the message to the dictionary
            # to then transform into a dataframe
            msg_iters[i] = (a,''.join([x for x in msg]))
        # if condition is false
```

```
        else:
            # apply the f function to the message instead
            for k in range(len(msg)):
                msg[k] = f(msg[k])
        # reset the message
        msg = list(message)

        # break the loop if 2 words are found in the english language corpus
        try:
            msg_list = msg_iters[i].split(' ')
            fw = {'w1':np.random.choice(msg_list),
                  'w2':np.random.choice(msg_list)}
            conds = {w_n:(w in words.words()) for w_n,w in fw.items()}
            vals = list(conds.values())
            if False not in vals:
                print(f'found at iteration: {i}')
                print(msg_iters[i])
                print(f'words found: {list(conds.keys())}')
                break
        # otherwise continue
        except:
            continue
    # put the information in a dataframe, iters, score and the messages
    df = {'iter':[it for it in msg_iters.keys()],
          'score':[msg[0] for msg in msg_iters.values()],
          'msg':[msg[1] for msg in msg_iters.values()]}
    # return the dataframe
    return pd.DataFrame(df)

# we run the function
result = decode(message,freq, 50000)
plt.plot(result.sort_values('score')['score'].reset_index(drop=True))
plt.show()
```




```
print(result[result['score'] == max(result['score'])])
#>         iter     score                                              msg
#> 30938  31353  1.319558  v svt vwfoc hcqfvqp hozvdo flo wqcbf mobfocp z...
```

```
#> [1] "n bnq ndrsw awornoc aslnxs rzs dowfr ksfrswc ljicrwq rj nirzjwoms n ljwjcngowif gnllocs aworofz
```

Problem 2

(a)

Let $N(t)$ be the number of cars arriving at a parking lot by time t, according to the proposed scenario, we can model $N(t)$ as a non-homogenous Poisson process. Such process has almost the same process as any other Poisson process, however, its rate is a function of time.

$N(t), t \in [0, \infty)$ is the non-homogenous Poisson process with rate $\lambda(t)$ where:

- $N(0) = 0$
- $N(t)$ has independent increments

We define 8:00 as $t = 0$ with the following integrable function and each unit of t equals to 1 hour:

$$\lambda(t) = \begin{cases} 100 & 0 \le t \le \frac{1}{2} \\ 600t - 200 & \frac{1}{2} < t \le \frac{3}{4} \\ 400t - 50 & \frac{3}{4} < t \le 1 \\ -500t + 850 & 1 < t \le 1.5 \end{cases}$$

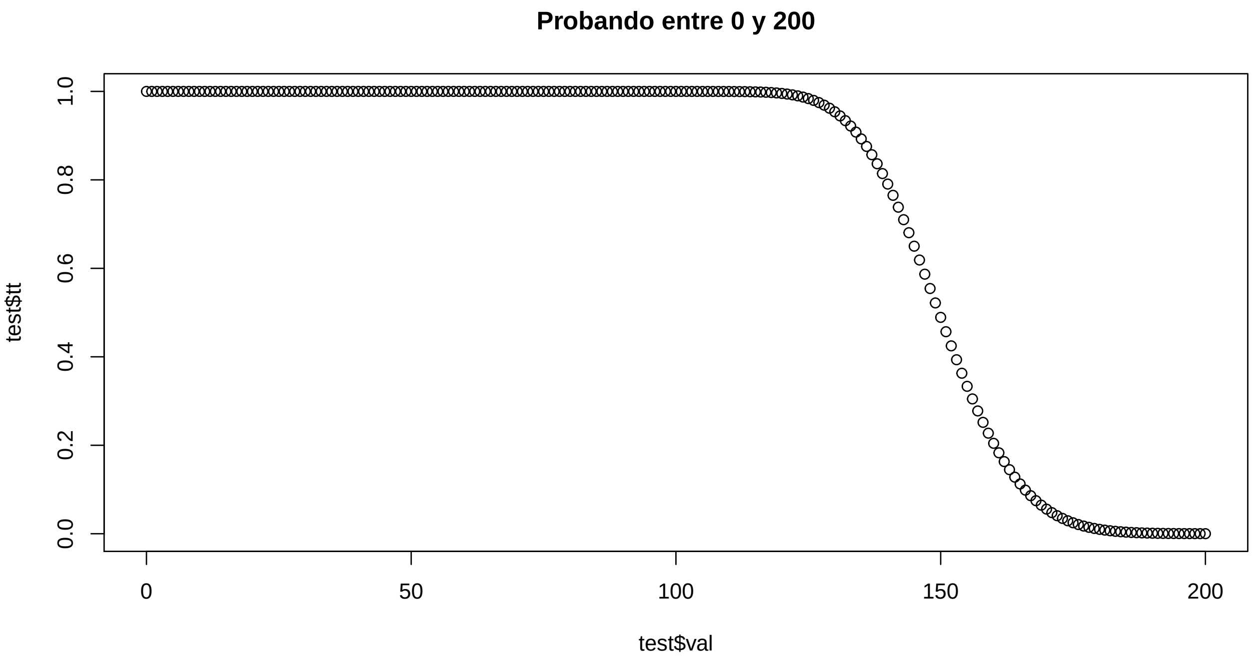
So,

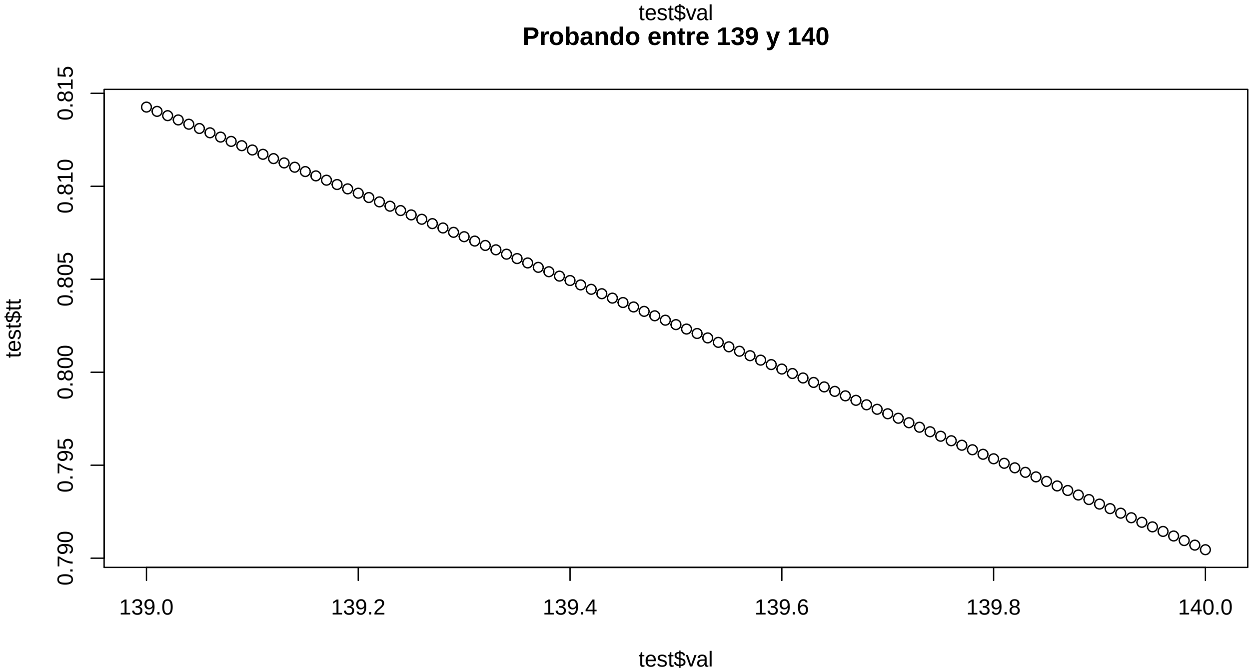
$$E[N(t)] = \begin{cases} \int_0^t 100\,dt = 100t & 0 \le t \le \frac{1}{2} \\ \int_{\frac{1}{2}}^t 600t - 200\,dt + 50 = 300(t^2 - \frac{1}{4}) - 200(t - \frac{1}{2}) + 50 & \frac{1}{2} < t \le \frac{3}{4} \\ \int_{\frac{3}{4}}^t 400t - 50\,dt + 93.75 = 25(8t^2 - 2t - 3) + 93.75 & \frac{3}{4} < t \le 1 \\ \int_1^t -500t + 850\,dt + 168.75 = -50(5t^2 - 17t + 12) + 168.75 & 1 < t \le 1.5 \end{cases}$$

Given that there is a limit of 150 vehicles:

$$E[N(t)] = \begin{cases} 100t & 0 \le t \le \frac{1}{2} \\ 300(t^2 - \frac{1}{4}) - 200(t - \frac{1}{2}) + 50 & \frac{1}{2} < t \le \frac{3}{4} \\ 25(8t^2 - 2t - 3) + 93.75 & \frac{3}{4} < t < 0.94468 \\ 150 & t \ge 0.94468 \end{cases}$$

(b)





Luego de hacer las pruebas para $\lambda(t)$ obtenemos lo siguiente:

```
lambda = 139.6
t = 0.91232
# 8:44 AM
```

Por lo que $t = 0.91232$ horas (aproximadamente a las 8:54 de la mañana).

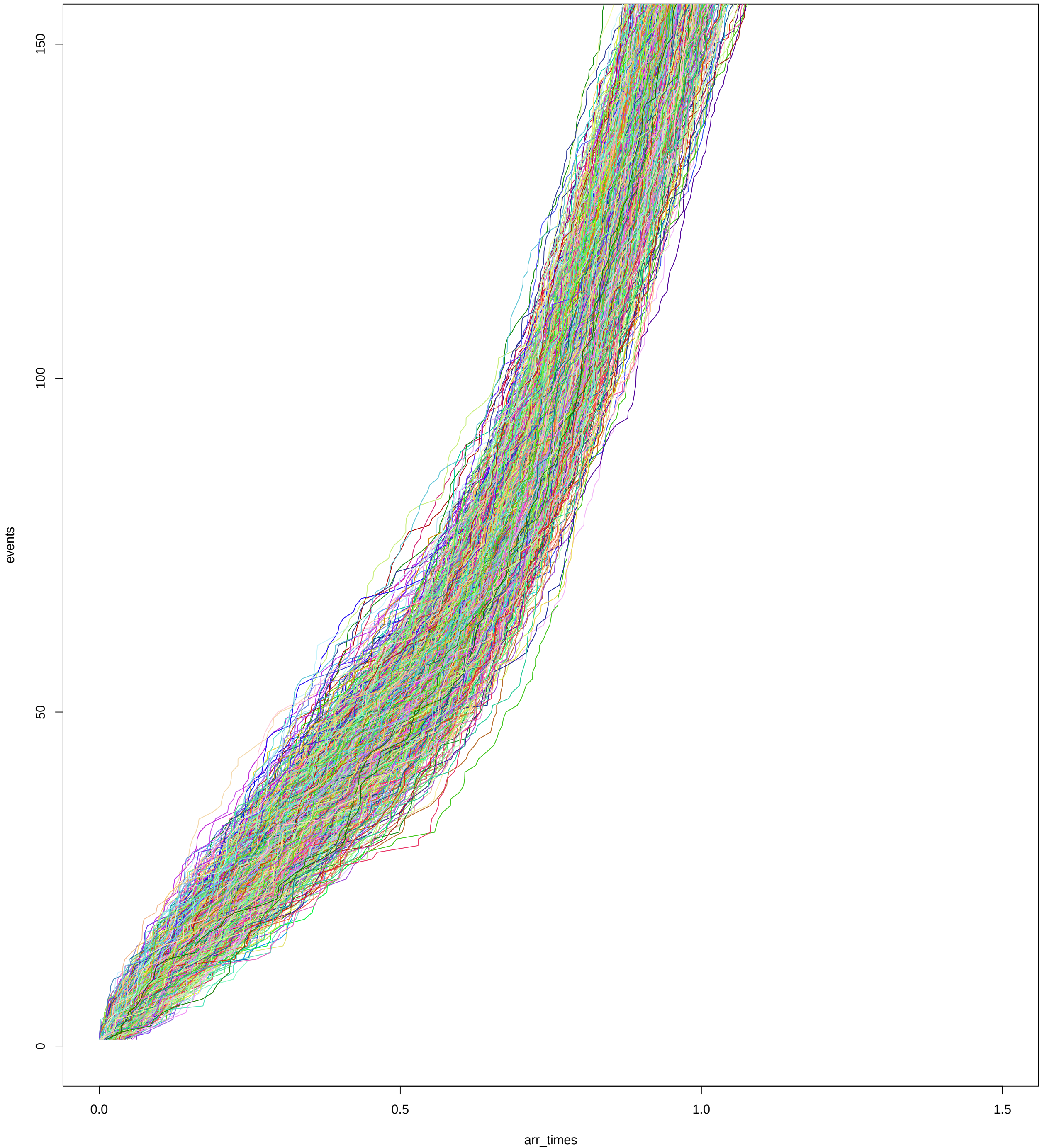
(c)

The following function simulates a non-homogenous poisson process from a homogenous poisson process:

```r
non_hom_poisson <- function(fun,l,a,b,start=0) {
    # This function generates a non-homogenous poisson
    # process from a homogenous poisson process
    # PARAMS:
    # fun:   if the non-homogenous poisson process has
    #        multiple functions per time subinterval
    #        this parameters represents such function
    # l:     lambda for the homogenous poisson process
    # a:     lower bound for the time subinterval
    # b:     upper bound for the time subinterval
    # start: this parameter is used to keep track of
    #        the process count.

    # We generate the homogenous poisson process
    # arrival times
    val <- rpois(1,l*(b-a))
    intervals <- (b-a) * sort(runif(val)) + a

    # Non-homogenous poisson process
    evs <- length(intervals) # lenght of arrival times
    nh_val <- 0 + start # start of the event count
    nh_intervals <- c() # arrival times for the NHPP
    for (i in 1:evs) {
        if (runif(1) < fun(intervals[i])/l) {
            # only including intervals from the HPP which
            # match with fun(intervals[i])/l probability
            nh_intervals <- c(nh_intervals, intervals[i])
            nh_val <- nh_val+1 # adding one to the event count
        }
    }
    nh_events <- seq(1+start,nh_val,1) # events since the previous group
    return(list(arrival_times=nh_intervals, events=nh_events))
}
```

```r
simulation <- function(iters, functions, lambdas, ints) {
    # This function simulates from the NHPP
    # iters:     number of iterations to plot and add to the list of
    #            dataframes
    # functions: list of functions corresponding to the lambda function
    # lambdas:   list of lambdas for each subinterval
    # ints:      lists of vectors of 2 elements each containing the intervals
    #            that correspond to each element of lambdas and functions lists
    p <- list()
    for (i in 1:iters) {
        maximum <- 0 # start for the next NHPP simulation to continue count
        arr_times <- c() # arrival times
        events <- c() # event counts
        for (k in 1:4) {
            int <- non_hom_poisson(lambda_funs[[k]],lambdas[[k]],
                                   ints[[k]][1],ints[[k]][2],
                                   start=maximum)
            maximum <- max(int$events) # remembering last event count
            arr_times <- c(arr_times, int$arrival_times)
            events <- c(events, int$events)
        }
        p[[i]] <- data.frame(arrival_times=arr_times, events=events)
        # plots
        if (i == 1) {plot(arr_times, events, cex=0.5, pch='.',
                          col=randomColor(), xlim=c(0,1.5),
                          ylim=c(0,150))}
        lines(arr_times, events, col=randomColor())
    }
    return(p)
}
```

```r
data <- simulation(1000, lambda_funs, lambdas, ints)
```




```r
ratio <- 0
for (i in 1:length(data)) {
    df <- data.frame(data[[i]])
    cnt <- df %>% filter(arrival_times < 0.91232 & events >= 150) %>% dplyr::count()
    if (cnt[1] >= 1) {
        ratio <- ratio + 1
    }
}
```

```r
ratio/1000
#> [1] 0.192
```

Problem 3

(a)

Our infinitesimal generator is the following:

$$Q = \begin{pmatrix} -\lambda & \lambda & 0 & 0 & \dots \\ \mu & -(\lambda+\mu) & \lambda & 0 & \dots \\ 0 & 2\mu & -(2\mu+\lambda) & \lambda & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

(b)

We solve the following system:

$$\begin{cases} \sum_{i=1}^{\infty} \pi_i = 1 \\ \lambda\pi_0 = \mu\pi_1 \\ \lambda\pi_1 = 2\mu\pi_2 \\ \vdots \\ \lambda\pi_{n-1} = 2\mu\pi_n \\ \vdots \end{cases}$$

First we have:

$\pi_1 = \frac{\lambda\pi_0}{\mu}$
$\pi_2 = \frac{\lambda^2\pi_0}{2\mu^2}$
$\pi_3 = \frac{\lambda^3\pi_0}{2^2\mu^3}$
$\vdots$
$\pi_n = \frac{\lambda^n\pi_0}{2^{n-1}\mu^n}$
$\vdots$

Then:

$\sum_{i=0}^{\infty} \pi_i = \pi_0 + \frac{\lambda\pi_0}{\mu} + \frac{\lambda^2\pi_0}{2\mu^2} + \dots + \frac{\lambda^n\pi_0}{2^{n-1}\mu^n} + \dots = 1$

And so factoring π_0 we get:

$\pi_0(1 + \frac{\lambda}{\mu} + \frac{\lambda^2}{2\mu^2} + \dots) = \pi_0(1 + \sum_{i=1}^{\infty} \frac{1}{2^{i-1}}(\frac{\lambda}{\mu})^i)$

Then multiplying $\frac{2}{2}$ to the summation:

$\pi_0(1 + \frac{2}{2}\sum_{i=1}^{\infty} \frac{1}{2^{i-1}}(\frac{\lambda}{\mu})^i)$

$= \pi_0(2\sum_{i=0}^{\infty}(\frac{\lambda}{2\mu})^i - 1)$

$\pi_0(2(\frac{1}{1-\frac{\lambda}{2\mu}}) - 1) = 1$

$\pi_0 = \frac{1}{2(\frac{1}{1-\frac{\lambda}{2\mu}})-1}$

$\vdots$

$\pi_n = \frac{\lambda^n}{2^{n-1}\mu_n\pi_0}\frac{1}{2(\frac{1}{1-\frac{\lambda}{2\mu}})-1}$

finally:

$\pi_n = \frac{1}{2^{n-1}}(\frac{\lambda}{\mu})^n\pi_0$

The infinite sum converges when $|\frac{\lambda}{2\mu}| < 1$ in which case the stationary distribution P exists.

Then:

$L = \sum_{n=0}^{\infty} \pi_n * n$

Using the following sum:

$\sum_{i=0}^{n-1} ia^i = \frac{a-na^n+(n-1)a^{n+1}}{(1-a)^2}$

As n approaches infinity:

$\sum_{i=0}^{\infty} ia^i = \frac{a}{(1-a)^2}$

We get the following:

$L = \pi_0[1 + \sum_{n=0}^{\infty}(\frac{\lambda}{\mu})^n * \frac{n}{2^{n-1}}]$

$L = \pi_0[1 + 2\sum_{n=0}^{\infty}(\frac{\lambda}{2\mu})^n * n]$

$L = \pi_0[1 + 2\frac{\frac{\lambda}{2\mu}}{(1-\frac{\lambda}{2\mu})^2}]$

(c)

Let's consider the probabilities conditioned on the number of customers in the system that are present once our specific subject l gets into the system.

If there are no other customers when l gets into the system, there is no chance of overtaking.

$P(N^{OV} = 0|N^{PR} = 0) = 1$

With N^{OV} being the number of customers that l overtakes and N^{PR} the number of customers present in the system (queing) when l gets in the system.

If $N^{PR} = 1$, then l can overtake only 1 customer, if the time it takes to be served is shorter than the time it takes the other customers to be served. Because of the memoryless property we can assert the following:

$P(N^{OV} = 0|N^{PR} = 1) = \frac{\mu}{\mu+\mu} = \frac{1}{2}$

Actually, in general:

$P(N^{OV} = k|N^{PR} = n) = \frac{1}{n+1}$, $n \leq c-1$, $x = 0, 1$

As in this case c=2, our l subject can' t overtake more than one customer.

Now, if $n \geq c$, that is, l has to get in queue and wait to be served. When l gets served, there is also one more customer getting served. Because, again, of the memoryless property.

$P(N^{OV} = k|N^{PR} = n) = \frac{1}{c}$, $n = c$, $k = 0, 1$

In our case, it does not matter how many customers are in the system, the probability of overtaking, conditioned to the number of customers already in the system, is $\frac{1}{2}$.

Now, using Bayes' theorem and the total probability rule, we can find the probability of l overtaking another customer.

$P(A|B) = \frac{P(A \cap B)}{P(B)}$

$\frac{1}{2}\sum_{i=1}^{\infty} \pi_i = \frac{1}{2}\sum_{i=1}^{\infty}(\frac{1}{2^{i-1}})(\frac{\lambda}{\mu})^i \pi_0$

$= \sum_{i=1}^{\infty}(\frac{\lambda}{2\mu})^i \pi_0 = (\sum_{i=0}^{\infty}((\frac{\lambda}{2\mu})^i) - 1)\pi_0$

$= (\frac{1}{1-\frac{\lambda}{2\mu}} - 1)\pi_0 = \frac{1}{2(\frac{1}{1-\frac{\lambda}{2\mu}-1})}(\frac{1}{1-\frac{\lambda}{\mu}} - 1)$

$= \frac{1}{1-\frac{\lambda}{2\mu}} - 1 = \frac{1-(1-\frac{\lambda}{2\mu})}{1-\frac{\lambda}{2\mu}} = \frac{\frac{\lambda}{2\mu}}{1-\frac{\lambda}{2\mu}}$

$= 2(\frac{1}{1-\frac{\lambda}{2\mu}}) - 1 = \frac{2}{1-\frac{\lambda}{2\mu}} - 1 = \frac{2-(1-\frac{\lambda}{2\mu})}{1-\frac{\lambda}{2\mu}}$

$= \frac{1+\frac{\lambda}{2\mu}}{1-\frac{\lambda}{2\mu}}$

Then:

$\frac{\frac{\lambda}{2\mu}}{1\frac{\lambda}{2\mu}} = \frac{\frac{\lambda}{2\mu}}{\frac{2\mu+\lambda}{2\mu}} = \frac{\lambda}{2\mu+\lambda}$

So then we get:

$P(N^{OV} = k) = \frac{\lambda}{2\mu+\lambda}$, $k = c - 1 = 1$

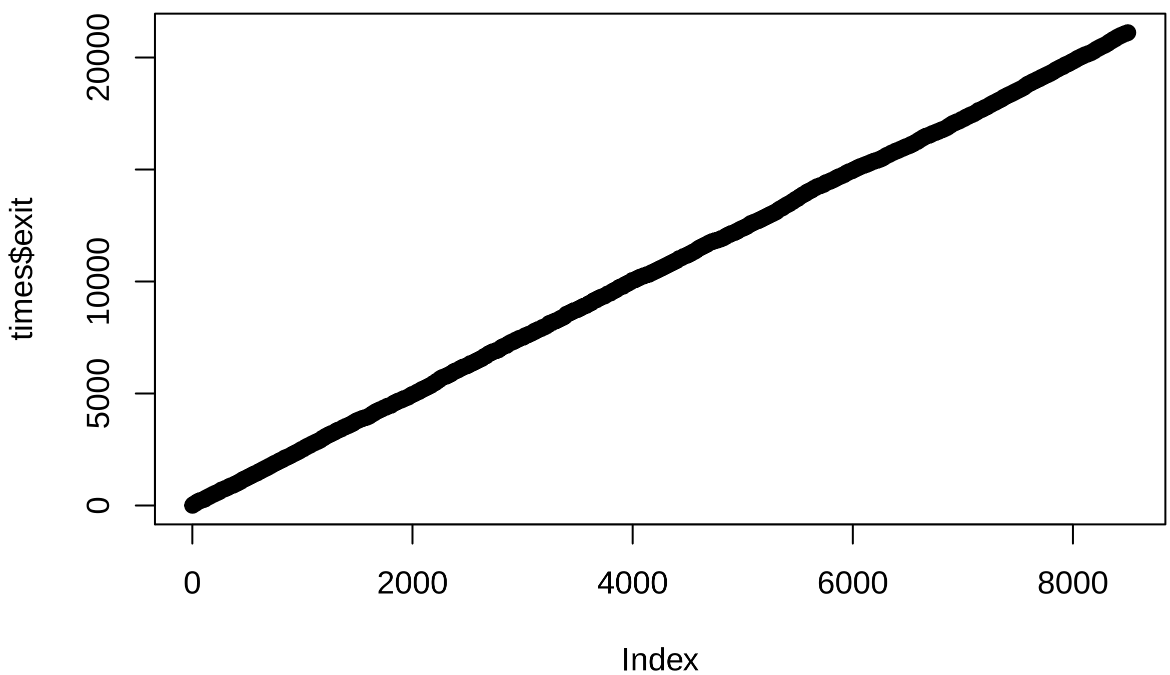
(d)

We define the following function to simulate the queueing system:

```r
# Simulation of the System (M/M/2)
q <- function(customers, l, m) {
  # This function generates a M/M/2 queue system and returns a matrix
  # with columns of ArrivalTimes, exit=ExitTimes and service=ServiceTimes
  # PARAMS:
  # customers:   number of customers in the supermarket
  # l:      lambda for the homogenous poisson process
  #    (customers arrive to the unique cashiers waiting
  #    line according this rate)
  # m:      mu for the exponential distribution
  #    (The times to be served are independent and
  #    distributed as exponential with rate mu)

  # interval for arrival times
  exp_at <- rexp(customers,l)

  # we have as many arrival times as the number of customers
  ArrivalTimes <- rep(0,customers)

  ArrivalTimes[1] <- exp_at[1]
  for (i in 2:customers) {
    # arrival time = the previous arrival time + the interval between 2 customers
    ArrivalTimes[i] <- ArrivalTimes[i-1] + exp_at[i]
  }

  # service time distributed according to mu
  ServiceTimes <- rexp(customers, m)

  # we have as many exit times as the number of customers
```

```
  ExitTimes <- rep(0,customers)

  # the first two exit time is equal to the service time due to we have 2 cashiers
  ExitTimes[1:2] <- ServiceTimes[1:2]
  for (i in 3:customers) {
    # we sort exit time from larger to smaller
    SortedTimes <- sort(ExitTimes[1:(i-1)], decreasing=T)

    # all of the two cashiers are occupied, then the new customer will have to
    # wait until at least one of them leaves the supermarket (the faster one)
    # then the exit time of the new customer is exited time of the previous faster customer plus
    # the service time of this new customer
    if (ArrivalTimes[i] < SortedTimes[2]) {ExitTimes[i] <- SortedTimes[2] + ServiceTimes[i]}

    # one or two cashiers are free, then the exit time is the arrival time plus the service time
    else {ExitTimes[i] <- ArrivalTimes[i] + ServiceTimes[i]}
  }

  # create a matrix and return it
  times <- data.frame(arrival=ArrivalTimes,
                      exit=ExitTimes,
                      service=ServiceTimes)
  return(times)
}
```

```
number_of_customers <- 8500
times <- q(number_of_customers, l=0.4, m=0.25)
plot(times$exit)
```



(e)

We define the following function to calculate the overtaking probability:

```r
d <- function(customers, queue, ExitTimes) {
  # This function generates a M/M/2 queue system and returns
  # the probability of overtaking
  # PARAMS:
  # customers:   number of customers in the supermarket
  # queue:       number of people in the quene
  # ExitTimes:   the exited times of each customer

  # we define the number of people of overtaking
  ot <- 0
  for (i in queue:(customers - queue)) {

    # the number of people overtaking of all customers is the previous
    # number plus 1 if the exit time of the i-th customer is smaller (earlier)
    # than the (i-1)th customer
    ot <- ot + sum(ExitTimes[i] < ExitTimes[1:(i-1)])
  }
  r <- customers-2*queue
  ot_prob <- ot/r
  return(ot_prob)
}
```

Write theRcode necessary to simulate the system (provide the code) and generate thetimes customers leave the supermarket

```r
avgs <- c() # general average of multiple simulations
number_of_customers <- 8500 # simulations done with 8500 people
for (i in 1:10) {
    times <- q(number_of_customers, l=0.4, m=0.25) # running the simulation
    avg <- c() # estimation (usually includes an extra person)
    avg_p <- c() # pessimistic estimation (usually excludes that extra person)
    for (i in 0:99) {
        # assuming the person number number_of_customers - i is the last one
        last_arrival_time <- times$arrival[length(times$arrival)-i]
        # checking all exit times mayores al arrival time of the person number number_of_customers - i
        val <- sum(times$exit[1:(length(times$arrival)-i-1)] > last_arrival_time)
        avg <- c(avg,val)
        avg_p <- c(avg,val-1)
    }
    # joining all estimations
    total_avg <- mean(c(avg,avg_p))
    # adding to vector with  all averages
    avgs <- c(avgs, total_avg)
}
# printing averages
avgs
#>  [1]  2.278607  4.228856  2.636816 11.771144  2.064677  3.621891  2.696517
#>  [8]  5.905473  2.179104 10.149254
# calculating the average of averages
mean(avgs)
#> [1] 4.753234
```

Calculation by hand:

```r
lambda <- 0.4
mu <- 0.25
lambda2mu <- lambda/(2*mu)

pi_0 <- 1/(2*(1/(1-lambda2mu)) - 1)
L <- pi_0 * (1 + 2*(lambda2mu/((1-lambda2mu)^2)))
```

$\frac{\lambda}{2\mu} = \frac{0.4}{2(0.25)} = 0.8$

$\pi_0 = \frac{1}{2(\frac{1}{1-0.8})-1} = \frac{1}{9} =\approx 0.\bar{1}$

$L = \pi_0(1 + 2\frac{0.8}{(1-0.8)^2}) = 4.\bar{5}$