Stochastic Processes: Assignment 1

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

January 10th, 2020

Problem 1

This is our implementation of the decoding algorithm, it is implemented using python. Every single step is explained during the code with its corresponding block or line(s) of code.

```
# 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, check_words=True):
    """
    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
    check_words = if true the loop will be broken if 5 words
                  in the message are found to be in the english
```

```
                language corpus, if false then the function will
                print the message after the condition where
                min(a(f*,f), 1) > random uniformly distributed number between
                0 and 1
    """
    # 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 + freqs[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 = np.exp(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]))

            # we separate the message by words
            msg_list = msg_iters[i][1].split(' ')
            # we test 5 random words to see if they are actually words
            # present in the word corpus
            conds = [np.random.choice(msg_list) in words.words() for x in range(5)]
            # if check words == True
```

```
            if check_words == True:
                # we break the loop if the 5 true words are found
                if False not in conds:
                    print('Cipher used:\n')
                    print(f'{cd}')
                    print(f'\nfound at iteration: {i}\nMessage (may contain wrong letters):')
                    return msg_iters[i]
            # otherwise
            else:
                # we print the iteration number and message only
                # when the cipher changes
                print(f'iteration #{i}:\n')
                print(f'{msg_iters[i][1]}\n')
        # if condition is false
        else:
            # resetting the letters after a failed score comparison
            letters_n = deepcopy(letters)
            # 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)

    # 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 in case the loop was never broken
    return pd.DataFrame(df)
```

Our function seems to oscillate between a nearly fully decoded message where b is mapped to p and the opposite (when it's allowed to run without a break)

As a result we obtain the following decoded message (either decoding by hand the last few wrong letters in the cipher):

"*a day after britain became the first western country to authorize a coronavirus vaccine british and american officials bickered over which governments drug approval process was better leading scientists to warn that the debate could undermine public faith vaccine nationalism has no place in covid or other public health matters of global significance said a scientific adviser to the british government science has always been the exit strategy from this horrendous pandemic several top british lawmakers have also incorrectly cast the countrys split with the european union as the reason it authorized a vaccine first in fact britain remains under the blocs regulatory umbrella and was able to move more quickly because of an old law enabling it to make its own determinations in public health emergencies*"

The message is fully decrypted after a variable number of iterations, sometimes it takes 3,000 iters but other tiems it might take 60,000 iters.

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
- for any $t \in [0, \infty)$, we have:
  - $P(N(t+\Delta) - N(t) = 0) = 1 - \lambda(t)\Delta + o(\Delta)$
  - $P(N(t+\Delta) - N(t) = 1) = \lambda(t)\Delta + o(\Delta)$
  - $P(N(t+\Delta) - N(t) \geq 2) = o(\Delta)$

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

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

We define, for all $t > 0, N_t$ has a poisson distribution with mean:

$E(N_t) = \int_0^t \lambda x \, dx$

So,

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

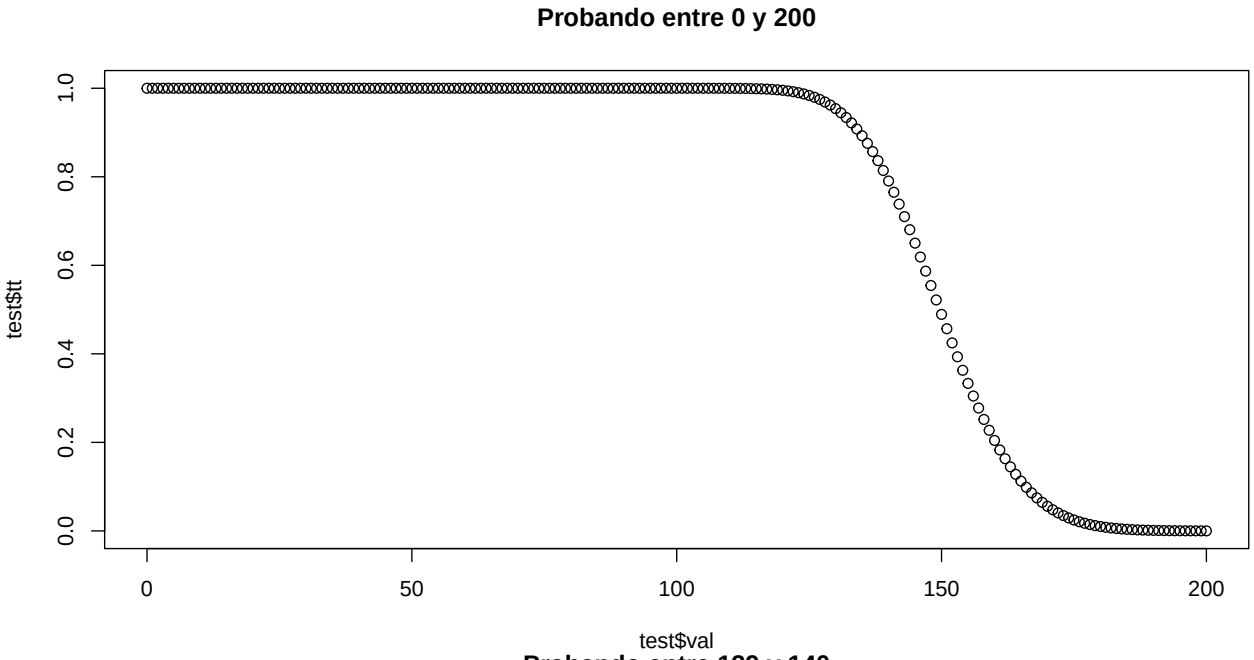
Given that there is a limit of 150 vehicles:

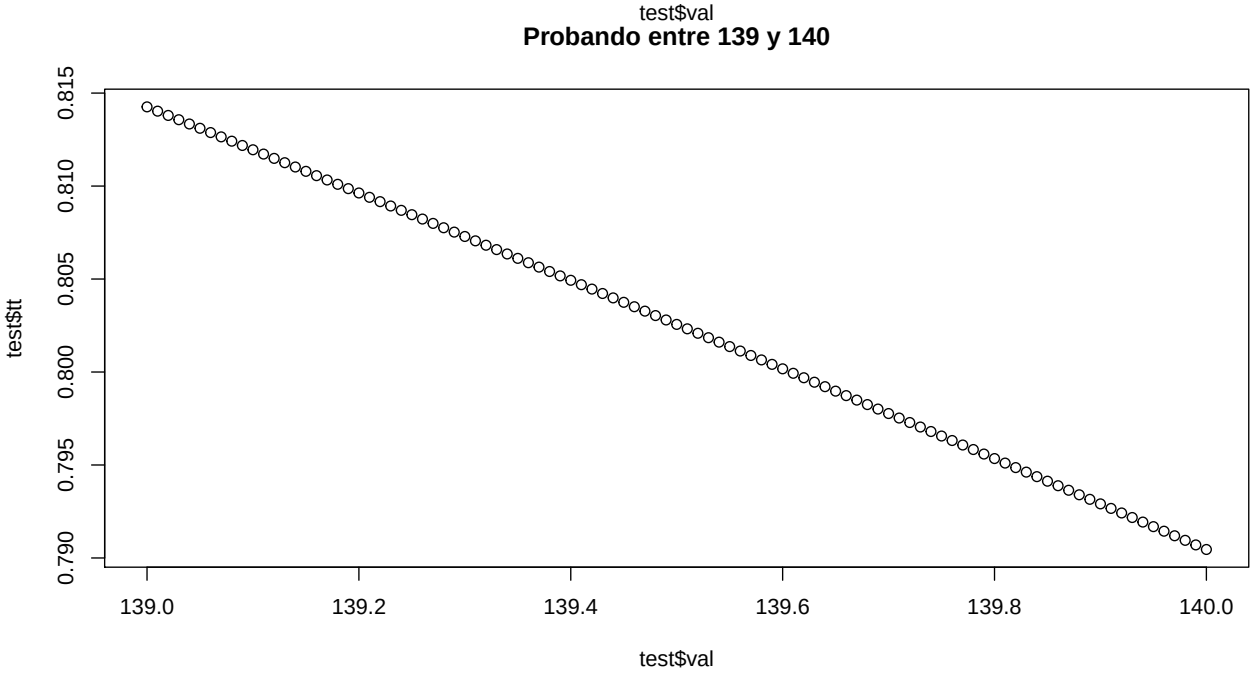
$$E[N(t)] = \begin{cases} 100t & 0 \leq t \leq \frac{1}{2} \\ 300(t^2 - \frac{1}{4}) - 200(t - \frac{1}{2}) + 50 & \frac{1}{2} < t \leq \frac{3}{4} \\ 25(8t^2 - 2t - 3) + 93.75 & \frac{3}{4} < t < 0.94468 \\ 150 & t \geq 0.94468 \end{cases}$$

(b)

We are asked the time in which, with a probability of 0.8, the number of vehicles in the parking lot is strictly less than 150. This way, we have a place in the parking. This is the same as saying that we want 0.8 to be the probability of having 149 or less cars in the parking. The function *ppois* in R, calculates (through the number of events and lambda) the probability of those events or less happening in the process. As we do have the probability, but are interested in finding lambda, what we do is try different values of lambda, until we get a probability equal to 0.8.

First, we try going unit by unit. When we know that it is between 139 and 140, we do the same but going in intervals of size 0.1. That is trying with values: 139.1, 139.2, and more.





Once we get lambda, with the expected value of events that we previously calculated, we tried calculating the times in the third interval.

As the time tested fits within the interval we are looking for, we know that it is the right one.

For example, if we use the first interval, the time calculated does not fit into its definition, and it would have been an error to consider this.

After testing the different $\lambda(t)$ we obtain the following:

$\lambda(t) \approx 139.6$ and $t \approx 0.91232$

Finally, we transform the result to a time format (as it is currently a decimal number representing hours) $t = 0.91232$ hours correspond approximately to 8:54 AM.

(c)

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

```
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))
}
```

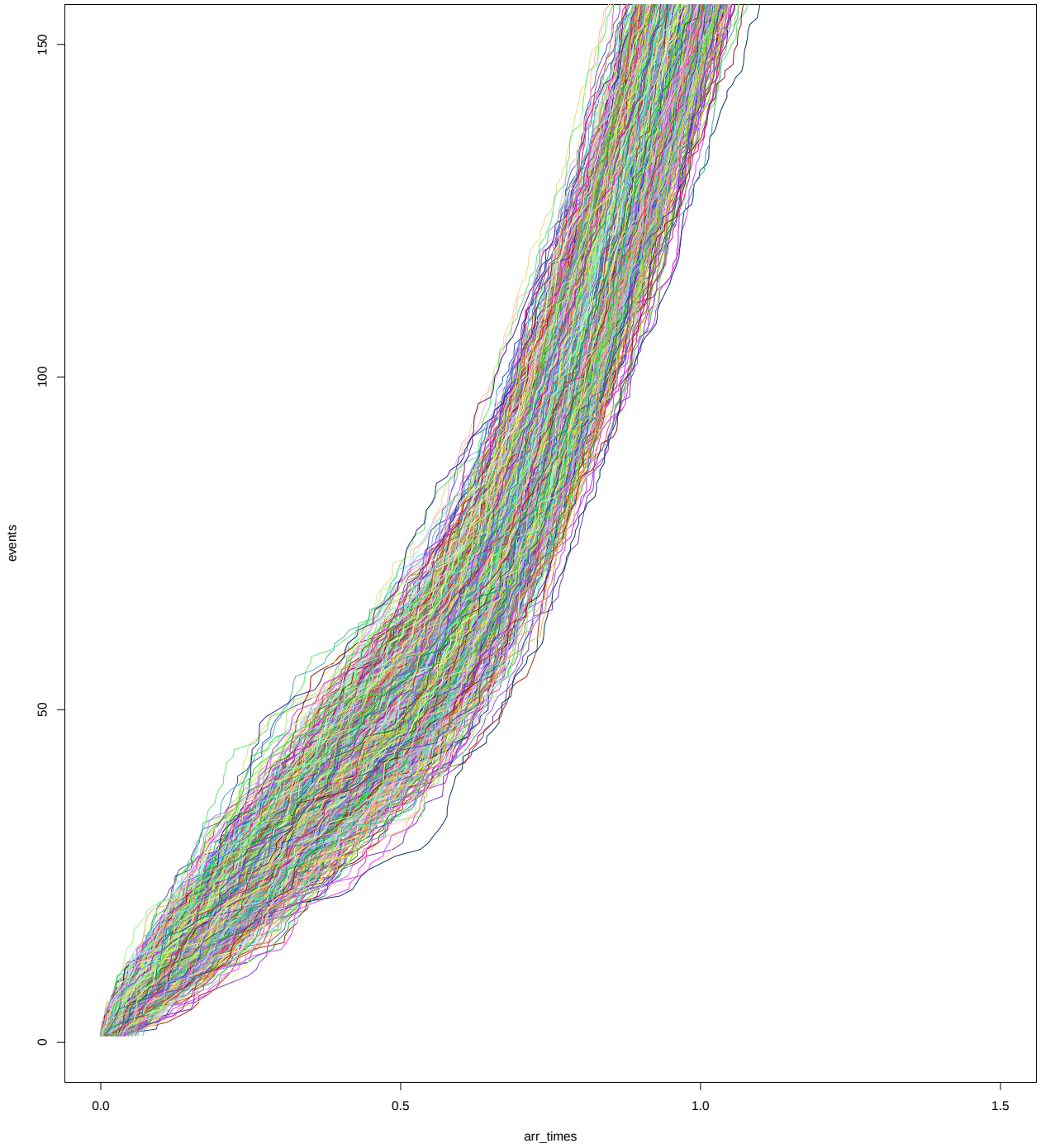
The following code iterates over *iters* amount of times the previously defined non-homogenous poisson process with the given function defining λ dependent on a parameter t ($\lambda(t)$).

```
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)
}
```

(d)

Running 1000 iterations of the simulation (we included the R output regardless because we thought it was very pretty).

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



Proving using the 1000 simulation data that our exercise 2b is correct:

```
#> [1] 0.206
```

We can see that our result approaches 0.2 very closely, which is the remaining probability ($1 - 0.8 = 0.2$).

Problem 3

(a)

An M/M/c queue is a stochastic process whose state space is the set {0, 1, 2, 3, ...} where the value corresponds to the number of customers in the system, including any currently in service. (Wikipedia: M/M/c queue)

Arrivals occur at rate λ according to a Poisson process and move the process from state i to *i+1*.

Service times have an exponential distribution with parameter μ. If there are fewer than 2 jobs, some of the servers will be free. If there are more than 2 jobs, the jobs queue in a buffer.

The buffer is of infinite size, so there is no limit on the number of customers it can contain.

In this case, the model describes a queue with two single servers which serves customers in the order in which they arrive. Customer arrivals constitute a Poisson process of rate λ.

This means that the number of customers arriving during a time period [t, t+s] is a Poisson random variable with mean λ, and it is independent of the past of the arrival process. Also, the inter-arrival times (between the arrival of one customer and the next) are identically distributed exponential random variables with parameter λ and mean $\frac{1}{\lambda}$. In other words, the arrival process is Markovian, and this is what the first M in M/M/2 stands for.

The second M of this process name says that the service process is Markovian. Customer service times are exponentially distributed, with parameter μ and mean $\frac{1}{\mu}$, and the service times of different customers are mutually independent. By the memoryless property of the exponential distribution, knowing how much service a customer has already received gives us no information about how much more service it requires. When a customer has been served, it departs from the queue. Also, in time 0, there are no costumers. So we see it fullfills all the conditions of a Markov Chain. (Ganesh 2012)

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:

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

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:

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

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

Knowing that this infite sum corresponds to a geometric series:

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

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

$\vdots$

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

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.

We can obtain Littles formula by finding the expectation with respect to the stationary distribution:

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

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:

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

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:

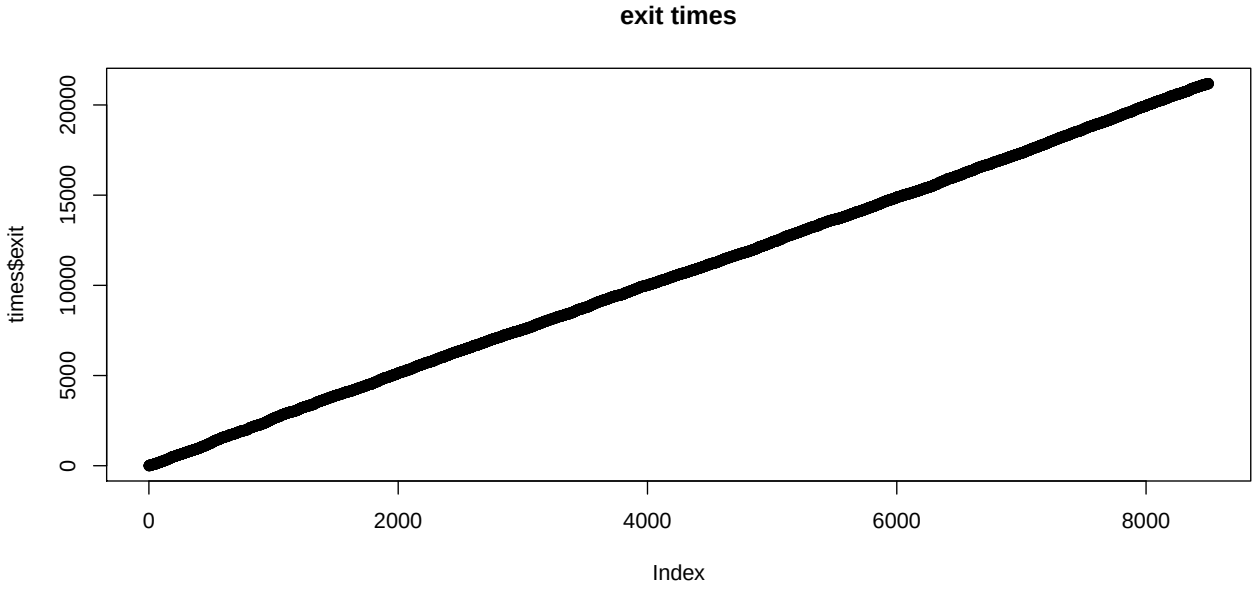
All the considerations and methodology are explained within the code in comments.

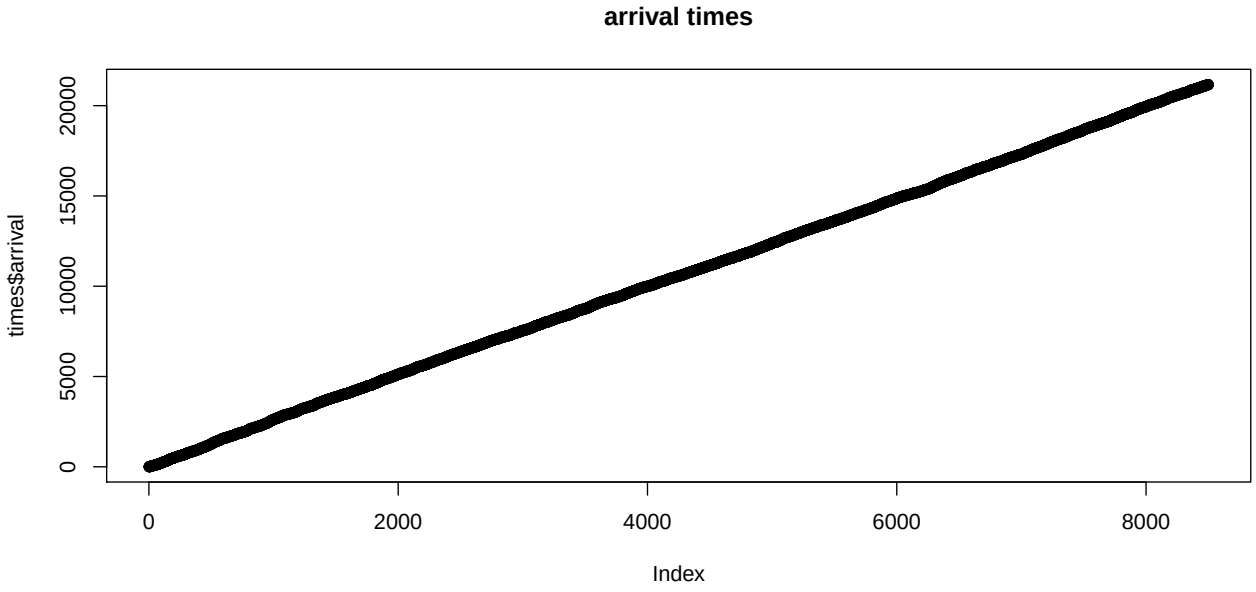
```
# Simulation of the System (M/M/2)
q <- function(customers, l, m) {
  # This function generates a M/M/2 queue system and returns a matrix
  # with columns of ArrivalTimes, exit=ExitTimes and service=ServiceTimes
  # PARAMS:
  # customers:   number of customers in the supermarket
  # l:     lambda 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)
}
```

The simulation uses the following params:

- number of customers = 8500
- $\lambda = 0.4$
- $\mu = 0.25$

We plot the cumulative sum of our exit and arrival times of our simulation:





(e)

1 - We define the following function to calculate the overtaking probability:

```
overtaking_prob <- 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)])
  }
  # we then calculate the probability
  r <- customers-2*queue
  ot_prob <- ot/r
  return(ot_prob)
}
```

Using the previous function to calculate the overtaking probability of our simulation:

```
#> [1] 0.4513333
```

We obtain an overtaking probability of ≈ 0.44.

To compare, we calculate the overtaking probability by hand:

$P(N^{OV}) = \frac{\lambda}{2\mu+\lambda} = \frac{0.4}{2*0.25+0.4} = 0.\bar{4}$

2- Estimating the long-run average of people in the system and comparing the result with part (b)

We first do so by simulating:

Our methodology is the following:

1 - We simulate 10 times using our simulation function for this exercises with 8500 customers

2 - We create an empty averages vector

3 - Within the loop we iteratively create vectors in order to add (to them) the number of customers in the system by assuming that the customer being iterated by is the last one

- if we have 8500 customers, we take the last 100 customers and we assume that the 8400 is the last one, then we assume the 8401 is the last one and so on.

4 - We continue doing this until we have a vector of 100 counts of customers in the system.

- While in the loop, we will create a vector with the exact count and the count minus one, so we have 2 vectors of counts, one with a pessimistic count (excluding the current customer that just came in) and one with an optimistic count (including the customer that just came in)

5 - We keep doing this until we have about 10 vectors (as we do 10 simulations) of 100 averaged vectors (optimistic and pessimistic counts of customers averaged per simulation) and we calculate the average of all the averages.

6 - We obtain the long-run average amount of customers in the system

```
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)
}
# calculating the average of averages
mean(avgs)
#> [1] 5.28408
```

Note: Given that this is a random process, different runs of the function we defined to do such simulation can yield different results. However, we found our results to be between 3 and 6.

Despite this, our most consistent results are around **4.4** and **4.5**.

Calculation by hand:

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

$\pi_0 = \frac{1}{2(\frac{1}{1-0.8})-1} = \frac{1}{9} = 0.\bar{1} \approx 0.11$

$L = \pi_0(1 + 2\frac{0.8}{(1-0.8)^2}) = 4.\bar{5} \approx 4.56$

References

Ganesh, Ayalvadi, Ph.D. 2012. "Simple Queueing Models."