55939 Stochastic Models

Final Project
The elevator queuing problem at Jerusalem's Yitzhak Navon train station

Ethan Zrahia, ID no.
Neta Bronfman, ID no.
Or Ben-Haim, ID no.

I. Background

This paper aims to find a solution for the queue waiting times for elevator service in Jerusalem's central train station Yitzhak Navon.

The Jerusalem Yitzhak Navon train station is the capital's central train station that opened just over four years ago. Construction of the station began in 2007 and was completed in 2018 at the cost of approximately US $140 million. Along with Jerusalem's central bus station right across the street, the Navon station constitutes a part of the main transportation hub of the city. A year after its opening in 2019, the Navon station was the 16th busiest train station in the country. In 2020, after additional construction making the direct Jerusalem to Tel Aviv service possible, the Navon station became the busiest train station, in the country outside of Tel Aviv. As of 2021 passenger traffic surpassed previous years', making Jerusalem's Navon station the fourth overall busiest train station in the country.

Due to Jerusalem's mountainous terrain, the train platforms had to be built 80 meters below street level, making the railway suitable for carrying passengers. The Navon station is the world's deepest heavy-rail passenger station and the fourth deepest underground station in the world. Because of the station's built, it can take anywhere from 5 to even 9 minutes to travel between the floors; this is based on our real measurements. This time doesn't include waiting lines in peak hours for the station's elevators, which we will go into more details later in the paper. This is significantly higher than the time it takes to reach platforms at other train stations. We believe this is a major bottleneck issue in commute time for Jerusalem's passengers going through the Navon station.

Currently, when passengers arrive (or leave) the station to (from) the train platforms, they have three options of doing so;

(1) Electric escalators: three different escalators must be taken to get to the platform level.

(2) Staircase: this option includes numerous flights of stairs and is the least used option by passengers.

(3) A system of three high-speed elevators.

While there is not much optimizing that can be done with the first two options, we will focus on the elevator's mechanism as a way to increase their efficiency and reduce waiting times.

An elevator's mechanism:

There are three different policies that can be applied to the elevator's mechanism, which are dictated by prioritizing a traffic flow[1].

1) **Up-peak:** this mechanism prioritizes a distinguished lobby floor where all passengers to arrive and request different floors. After an elevator takes a passenger to his desired floor, it rapidly returns to the lobby after delivering the passenger.
2) **Down-peak:** opposite mechanism of up-peak.
3) **Heavy-two-way:** doesn't prioritize a direction of traffic. If, for example, a passenger in the lobby requests to go to floor 6, the elevator will take him to his desired floor and will stay there after delivering the passenger, until it is ordered by a different one.

Elevator's mechanism in the Yitzhak Navon station

[1] Need a Lift? An Elevator Queuing Problem

The current situation in the Navon station is not optimal. Waiting times in queues for elevators on both floors (entrance and platforms) are long. Based on our real time measurements, passengers can wait anywhere from a minute to more than five minutes. This might be because of the elevators' current heavy-two-way traffic mechanism. If, for example, a train full of passengers arrive to the station and the button of the elevator is pressed, only one elevator will come down. Only after that elevator's door closes and it starts making its trip to the desired floor, a second elevator will come down. This leaves a large group of passengers waiting and significantly increases waiting times.

Possible solution

We believe the elevator's current mechanism (heavy two-way) is the possible issue, and there lies the solution. Our simulation will check which mode leads to the lowest waiting time.

II. Literature background

We first base our simulation off queuing theory taught throughout the course. Although it is not the exact queue type in our case, the M/D/1 queue provides the closest most accurate representation of the queues in the Navon station. There are two main points of difference between the M/D/1 queue and the queues we will show in the simulation. First, showcasing the two queues (one at each floor) as independent of one another is inaccurate; in reality, both queues at the different floors depend on the same server and so each queue's behavior and interarrival rate affects the other queue. Second, the M/D/1 queue theory we will show assumes the passengers arrive in batches of 16 people, which is the maximum capacity of the elevator. We also assume this as a way to simplify, but it is not an accurate representation that models the real arrival, which isn't necessarily in batches.

Having said that, we will present the M/D/1 queuing theory as theoretical background and a benchmark, a point of comparison to the simulation that will be presented afterwards.

Queuing Theory (applies to both queue A and B)

Queue A: the queue created at floor 0 (train platform underground floor) of the passengers that disembark the train and are looking to go up to floor 1 (entrance ground floor) and exit the station. Demand for this queue is represented by *n_down* in the simulation.

Queue B: the queue created at floor 1 (ground floor) of passengers arriving to the train station and looking to go down to floor 0 (underground floor) to board the train. Demand for this queue is represented by *n_up* in the simulation.

- **Queue type: M/D/1**
  - **M** – exponential interarrival (Poisson) time distribution with Markov properties (stochastic).
    - The arrivals of passengers occur at different times, and the interarrival times are usually considered independent from one another. We consider the arrival times to be exponentially distributed, and the arrival flow will be a Poisson process for both queues.
    - We view the arrivals of passengers as arrivals of **batches** of passengers. The batches are of 16 people, which is the maximum capacity of the elevators at the station. This seems like a reasonable assumption to us because this fits the typical arrival behavior of passengers to a train station; for queue A the train unloads a group of people, or multiple batches, at the same time. As for queue B, people will generally arrive at a similar time to depart a train that leaves at a certain time. Also, the server serves batches of people at a time. This allows us to use the M/D/1 queue. It is an approximate assumption that because passengers arrive in batches, the interarrival times are a Poisson distribution.[2]
    - The λ value is different for each queue.
  - **D** – deterministic service time.
    - In our case of one-floor traffic (two floors total) the service time is deterministic and not exponential.[3] This is because there are

[2] Stochastic Models in Queuing Theory, Second Edition, 2003
[3] Need a Lift? An Elevator Queuing Problem

no stops of loading and unloading; hence, service time from start to finish is uninterrupted.

- We define service time as the time it takes for the elevator to serve the passenger: take him to his desired floor and make the trip back.
- In the simulation: service time = 1 minute (1/60 of an hour). This is our real time measurements from testing out the elevators in the Navon station.
- If service time is constant and equals x and $x = s$, then:
    - $\bar{x} = s$
    - $\overline{x^2} = s^2 = \bar{x}^2 \rightarrow var(x) = \overline{x^2} - \bar{x}^2 = 0$

- **1** – number of servers (elevators).
    - To simplify the simulation, we assume there is one elevator instead of three. We believe the simulation's results provide close enough results with one server.

Comparing theory to simulation:

We are using the **Pollaczek–Khinchine** formula to look at the relationship between waiting times and rate of passengers' arrival λ.

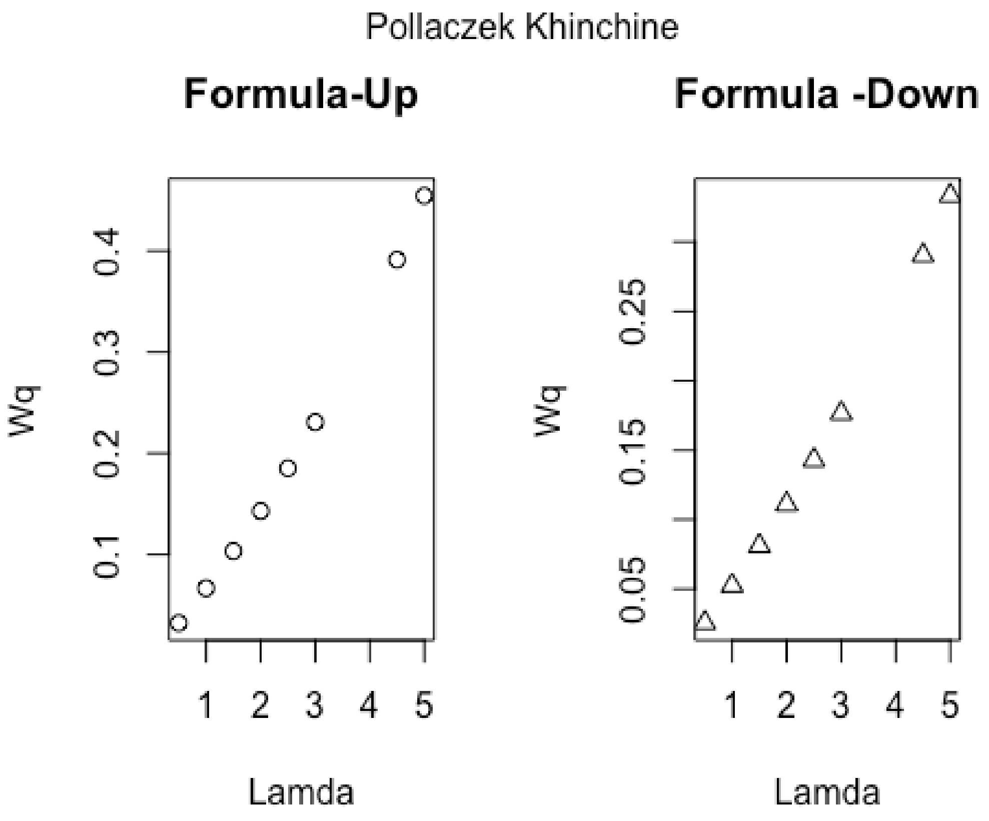
- ➔ Since service time is deterministic and var(s) = 0
- ➔ The ASTA property holds, so we can assume $V_q = W_q$
- ➔ The interarrival rate is Poisson, so the PASTA property also holds.
- ➔ The Pollaczek-Khinchine formula for an M/D/1 queue:

$$W_q = \left(\frac{\rho}{2(1-\rho)}\right) \cdot \bar{x}$$

- ➔ While $\rho = \lambda \cdot \bar{x}$
- ➔ $\bar{x} = 2$ and is deterministic. Although the real service time is 1 minute, we assume for the theory that both queues are independent of one another (not the case in reality), so we're also considering the elevator's trip back.

- ➔ Because the theory assumes the queues are independent, the different elevator's policies don't play a role in this analysis. They will come into place in the simulation.
- ➔ In the graphs below (calculation in excel sheet are attached) we are showing the relation between different λ and W_q :

```
x1<-c(0.5,1,1.5,2,2.5,3,4.5,5)
x2<-c(0.5,1,1.5,2,2.5,3,4.5,5)
y1<-c(0.0323,0.0667,0.1034,0.1429,0.1852,0.2308,0.3913,0.4545)
y2<- c(0.0256 ,0.0526 ,0.0811 ,0.1111,0.1429,0.1765,0.2903,0.3333)
par(mfrow=c(1,2))
plot(x1,y1, pch = 1,xlab = "Lamda",ylab = "Wq",main="Formula-Up")
plot(x2,y2, pch = 2,xlab = "Lamda",ylab = "Wq",main="Formula -Down")
mtext("Pollaczek Khinchine",side = 3,line = -1,outer = TRUE)
```



III. The simulation

Before testing out our simulation of the different elevator's policies, we will provide basic assumptions of the two queues' properties in the system, so we can make further conclusions later on. We base this model off the material taught throughout class and base some assumptions with the assistance of previous research. With the R simulation, we can reach a better approximation to real life than the theoretical queues. The previous section of this paper required us to make a few assumptions that aren't realistic, in order for it to fit the theoretical material. Having said that, we had to make some assumptions in the simulation framework as well, in order to simplify our model. The assumptions are as follows:

General information:

- The simulation analyzes two queues – one at the entrance floor (floor 1) and at the platform floor (floor 0).
- The Interarrival time distribution of passengers at the entrance floor is exponential with a constant λ throughout the whole simulation.
- The interarrival time distribution of passengers at the platform's floor is also exponential but with a changing λ ; At the beginning of the simulation the λ value is higher, and at the end it is lower. The exact time when the λ changes is not deterministic, but the expected value of the time of switch is after the simulation runs 1/3 of its total run.
- The simulation has one elevator (server) that goes between the two floors. Its service time is deterministic and equals 1 minute exactly. We constitute service time as loading of the passengers, opening and closing of the elevator doors, and traveling one way to the other floor.)

- The elevator has three different possible mechanisms (we also call this policies), and the policy stays constant with every loop that the simulation does (similar explanation is mentioned previously in the literature section):

    - Heavy two-way
    - Up-peak
    - Down-peak

- The maximum capacity of the elevator is 16 passengers.
- 1 run of the simulation is a <u>30-minute time frame</u>. It starts when a train full of passengers arrives to the station and ends when the next train departure takes place (the arrival times of the trains are deterministic).
- Unlike the M/D/1 queues we showed before which were independent of one another, in the simulation the two queues depend on the same server, and one is affected by the other.
- Although the system is not stable, we assume and use Little's Law.
- Queue discipline
    - The queue is a policy with no disruptions.
    - Queue discipline is FCFS, although it can't be noticed in the simulation. However, based on little's law and what we learned in class, it can be either FCFS or LCFS. The queue discipline does not affect waiting times.
- The capacity of the system is infinite; there is no limitation to the number of passengers that can wait in either queue, although the number of passengers looking to exit the train station will not exceed the passenger capacity of the train.

Simulation axioms:

- Memoryless property – as we mentioned, the assumption of the exponential interarrival time distribution means that we're dealing with a Poisson process. Hence, because of the memoryless property, the arrival time of the next passenger can be reset each time regardless of the previous arrival times.
- In addition, instead of the simulation scheduling the arrival of the passengers separately for each queue, we are calculating the arrivals <u>combined</u>, by adding the λ value of both queues.

- In order to optimize the simulation, the time clock in the simulation 'jumps' and basically skips over time slots where nothing is happening, and only counts times when something happens; for example, when a passenger arrives (we know which floor he arrived to by the λ rations) or when the elevator reached one of the floors.
- Calculating the average waiting time with Little's Law – we know the value of the average λ for each queue and the average length of both queues at any given moment. Therefore, we can use Little's Law and calculate W for each queue.

Running the simulation:

- We ran through the simulation 500 times for every elevator policy.
- The simulation gives back 4 results (by order from left to right): average waiting time at the entrance floor, average waiting time at the platform floor, the number of passengers left waiting at the entrance floor and the number of passengers left waiting at the platform floor.
- We repeat this process multiple times and each time are changing the λ value, the arrival rate of passengers.
- We made sure to maintain the ration between the two floors' λ value each time we ran the simulation, but each time we changed their absolute size (that is, the ration between the two rates remained the same, and each time we multiplied the lambda rates by a certain coefficient, as we will show in the graphs below).

Simulation:

```
# heavy two ways

two_ways <- function(lambda_high, lambda_low, lambda_up)
{
  counter <- 0
  n_up <- 0 # number of people up
  n_down <- 0 # number of people down
  l_train <- 0 # how much time passed since the last train
  elevator_state <- 0

  train_freq <- 30
  theta <- 1/10
  state1 <- 1


  elevator_travel <- 1
  elevator_capacity <- 16

  T <- 30

  clock <- 0

  L_up <- 0 # number of travelers in the system
  L_down <- 0
  avg_lambda_down <- (1/3)*lambda_high + (2/3)*lambda_low
  avg_lambda_up <- lambda_up
  elev_time <- 0
  spe_time <- 0
  last_clock <- 0
  res_up <- rep(NaN, 500)
  res_down <- rep(NaN, 500)
  res_nup <- rep(NaN, 500)
  res_ndown <- rep(NaN,500)
  for (i in 1:500)
  {
    while(clock<T) {
      if(n_up==0 & n_down==0){ #no one
        counter <- counter+1
        time <-rexp(1,lambda_up+lambda_high*state1+ lambda_low*(1-state1)+
theta)
        if (clock + time <= elev_time+1 || elev_time==0)
        {
          clock <- clock+time
          spe_time <- 0
        }
        else
        {
          spe_time <- 1
          clock <- elev_time+1
          elevator_state <- elevator_state + 0.5
          elev_time <- 0
        }
        L_up <- L_up + n_up*(clock-last_clock)
        L_down <- L_down + n_down*(clock-last_clock)
        if (spe_time==0 && clock>0)
        {
          up_or_down <- runif(1,0,1)
          if (up_or_down>(avg_lambda_up/(avg_lambda_up+avg_lambda_down)))
          {
            n_down <- n_down + 1
          }
          else
          {
            n_up <- n_up + 1
          }
```

```r
        }
      }
      else
      {
        if ((n_down>0) & (n_up>0)) # people in both queues
        {
          if (elevator_state==0)
          {
            if (n_down>=elevator_capacity)
            {
              n_down <- n_down-elevator_capacity
            }
            else
            {
              n_down <- 0
            }
            elevator_state <- 0.5
            elev_time <- clock

          }
          else
          {
            if (elevator_state==1)
            {
              if (n_up>=elevator_capacity)
              {
                n_up <- n_up-elevator_capacity
              }
              else
              {
                n_up <- 0
              }
              elevator_state <- -0.5
              elev_time <- clock
            }
          }
        }
        if (n_down==0 & n_up>0) # people only upstairs
        {
          if (elevator_state==1)
          {
            if (n_up>=elevator_capacity)
            {
              n_up <- n_up-elevator_capacity
            }
            else
            {
              n_up <- 0
            }
            elevator_state <- -0.5
            elev_time <- clock
          }
          else
          {
            if (elevator_state==0)
            {
              elevator_state <- 0.5
              elev_time <- clock
            }
          }

        }
        if (n_down>0 & n_up==0) # people only downstairs
        {
          if (elevator_state==0)
          {
            if (n_down>=elevator_capacity)
            {
              n_down <- n_down-elevator_capacity
            }
            else
            {
              n_down <- 0
            }
            elevator_state <- 0.5
            elev_time <- clock

          }
          else
          {
            if (elevator_state==1)
            {
              elevator_state <- -0.5
              elev_time <- clock
            }
          }
        }
        time <-rexp(1,lambda_up+lambda_high*state1+ lambda_low*(1-state1)+
theta)
        if (clock + time <= elev_time+1 || elev_time==0)
        {
          clock <- clock+time
          spe_time <- 0
        }
        else
        {
          spe_time <- 1
          clock <- elev_time+1
          elevator_state <- elevator_state + 0.5
          elev_time <- 0
        }
        L_up <- L_up + n_up*(clock-last_clock)
        L_down <- L_down + n_down*(clock-last_clock)
        if (spe_time==0 && clock>0)
        {
          up_or_down <- runif(1,0,1)
          if (up_or_down>(avg_lambda_up/(avg_lambda_up+avg_lambda_down)))
          {
            n_down <- n_down + 1
          }
          else
          {
            n_up <- n_up + 1
          }
        }
      }
      last_clock <- clock
    }
    W_up <- (L_up/T)/avg_lambda_up # average waiting time
    W_down <- (L_down/T)/avg_lambda_down
    res_up[i]= W_up
    res_down[i] <- W_down
    res_nup[i] <- n_up
    res_ndown[i] <- n_down
    counter <- 0
    n_up <- 0 # number of people up
    n_down <- 0 # number of people down
    l_train <- 0 # how much time passed since the last train
    elevator_state <- 0

    train_freq <- 30
    theta <- 1/10
    state1 <- 1

    elevator_travel <- 1
    elevator_capacity <- 16

    T <- 30

    clock <- 0

    L_up <- 0 # number of travelers in the system
    L_down <- 0
    elev_time <- 0
    spe_time <- 0
    last_clock <- 0


  }

  ret <- c(mean(res_up),mean(res_down), mean(res_nup), mean(res_ndown))
  return (ret)
}


##########################################################################
```

```r
##########################################################################
#up

up_peak <- function(lambda_high, lambda_low, lambda_up)
{
  counter <- 0
  n_up <- 0 # number of people up
  n_down <- 0 # number of people down
  l_train <- 0 # how much time passed since the last train
  elevator_state <- 0

  train_freq <- 30
  theta <- 1/10
  state1 <- 1


  elevator_travel <- 1
  elevator_capacity <- 16

  T <- 30

  clock <- 0

  L_up <- 0 # number of travelers in the system
  L_down <- 0
  avg_lambda_down <- (1/3)*lambda_high + (2/3)*lambda_low
  avg_lambda_up <- lambda_up
  elev_time <- 0
  spe_time <- 0
  last_clock <- 0
  res_up <- rep(NaN, 500)
  res_down <- rep(NaN, 500)
  res_nup <- rep(NaN, 500)
  res_ndown <- rep(NaN,500)
  for (i in 1:500)
  {
    while(clock<T) {
      #message ("clock= ",clock)
      #print(elevator_state)
      #print(n_up)
      #print(n_down)
      #print(L_down)
      if(n_up==0 & n_down==0){ #no one
        counter <- counter+1
        time <-rexp(1,lambda_up+lambda_high*state1+ lambda_low*(1-state1)+
theta)
        if (clock + time <= elev_time+1 || elev_time==0)
        {
          clock <- clock+time
          spe_time <- 0
        }
        else
        {
          spe_time <- 1
          clock <- elev_time+1
          elevator_state <- elevator_state + 0.5
          elev_time <- 0
        }
        L_up <- L_up + n_up*(clock-last_clock)
        L_down <- L_down + n_down*(clock-last_clock)
        if (spe_time==0 && clock>0)
        {
          up_or_down <- runif(1,0,1)
          if (up_or_down>(avg_lambda_up/(avg_lambda_up+avg_lambda_down)))
          {
            n_down <- n_down + 1
          }
          else
          {
            n_up <- n_up + 1
          }
        }
        if (elevator_state==0)
        {
          elevator_state <- 0.5
          elev_time <- clock
        }
      }
      else
      {
        if ((n_down>0) & (n_up>0)) # people in both queues
        {
          if (elevator_state==0)
          {
            if (n_down>=elevator_capacity)
            {
              n_down <- n_down-elevator_capacity
            }
            else
            {
              n_down <- 0
            }
            elevator_state <- 0.5
            elev_time <- clock

          }
          else
          {
            if (elevator_state==1)
            {
              if (n_up>=elevator_capacity)
              {
                n_up <- n_up-elevator_capacity
              }
              else
              {
                n_up <- 0
              }
              elevator_state <- -0.5
              elev_time <- clock
            }
          }
        }
        if (n_down==0 & n_up>0) # people only upstairs
        {
          if (elevator_state==1)
          {
            if (n_up>=elevator_capacity)
            {
              n_up <- n_up-elevator_capacity
            }
            else
            {
              n_up <- 0
            }
            elevator_state <- -0.5
            elev_time <- clock
          }
          else
          {
            if (elevator_state==0)
            {
              elevator_state <- 0.5
              elev_time <- clock
            }
          }

        }
        if (n_down>0 & n_up==0) # people only downstairs
        {
          if (elevator_state==0)
          {
            if (n_down>=elevator_capacity)
            {
              n_down <- n_down-elevator_capacity
            }
            else
            {
              n_down <- 0
            }
            elevator_state <- 0.5
            elev_time <- clock

          }
          else
          {
            if (elevator_state==1)
            {
              elevator_state <- -0.5
              elev_time <- clock
            }
          }
```

```r
        }
        time <-rexp(1,lambda_up+lambda_high*state1+ lambda_low*(1-state1)+
theta)
        if (clock + time <= elev_time+1 || elev_time==0)
        {
          clock <- clock+time
          spe_time <- 0
        }
        else
        {
          spe_time <- 1
          clock <- elev_time+1
          elevator_state <- elevator_state + 0.5
          elev_time <- 0
        }
        L_up <- L_up + n_up*(clock-last_clock)
        L_down <- L_down + n_down*(clock-last_clock)
        if (spe_time==0 && clock>0)
        {
          up_or_down <- runif(1,0,1)
          if (up_or_down>(avg_lambda_up/(avg_lambda_up+avg_lambda_down)))
          {
            n_down <- n_down + 1
          }
          else
          {
            n_up <- n_up + 1
          }
        }
      }
      last_clock <- clock
    }
    W_up <- (L_up/T)/avg_lambda_up # average waiting time
    W_down <- (L_down/T)/avg_lambda_down
    res_up[i]= W_up
    res_down[i] <- W_down
    res_nup[i] <- n_up
    res_ndown[i] <- n_down
    counter <- 0
    n_up <- 0 # number of people up
    n_down <- 0 # number of people down
    l_train <- 0 # how much time passed since the last train
    elevator_state <- 0

    train_freq <- 30
    theta <- 1/10
    state1 <- 1
    elevator_travel <- 1
    elevator_capacity <- 16

    T <- 30

    clock <- 0
    L_up <- 0 # number of travelers in the system
    L_down <- 0
    elev_time <- 0
    spe_time <- 0
    last_clock <- 0


  }

  ret <- c(mean(res_up),mean(res_down),mean(res_nup),mean(res_ndown))
  return (ret)

}



##########################################################################
#####



##########################################################################
#####
#Down

down_peak <- function(lambda_high, lambda_low, lambda_up)
{
  counter <- 0
  n_up <- 0 # number of people up
  n_down <- 0 # number of people down
  l_train <- 0 # how much time passed since the last train
  elevator_state <- 0

  train_freq <- 30
  theta <- 1/10
  state1 <- 1


  elevator_travel <- 1
  elevator_capacity <- 16

  T <- 30

  clock <- 0

  L_up <- 0 # number of travelers in the system
  L_down <- 0
  avg_lambda_down <- (1/3)*lambda_high + (2/3)*lambda_low
  avg_lambda_up <- lambda_up
  elev_time <- 0
  spe_time <- 0
  last_clock <- 0
  res_up <- rep(NaN, 500)
  res_down <- rep(NaN, 500)
  res_nup <- rep(NaN, 500)
  res_ndown <- rep(NaN,500)
  for (i in 1:500)
  {
    while(clock<T) {
      #message ("clock= ",clock)
      #print(elevator_state)
      #print(n_up)
      #print(n_down)
      #print(L_down)
      if(n_up==0 & n_down==0){ #no one
        counter <- counter+1
        time <-rexp(1,lambda_up+lambda_high*state1+ lambda_low*(1-state1)+
theta)
        if (clock + time <= elev_time+1 || elev_time==0)
        {
          clock <- clock+time
          spe_time <- 0
        }
        else
        {
          spe_time <- 1
          clock <- elev_time+1
          elevator_state <- elevator_state + 0.5
          elev_time <- 0
        }
        L_up <- L_up + n_up*(clock-last_clock)
        L_down <- L_down + n_down*(clock-last_clock)
        if (spe_time==0 && clock>0)
        {
          up_or_down <- runif(1,0,1)
          if (up_or_down>(avg_lambda_up/(avg_lambda_up+avg_lambda_down)))
          {
            n_down <- n_down + 1
          }
          else
          {
            n_up <- n_up + 1
          }
        }
        if (elevator_state==1)
        {
          elevator_state <- -0.5
          elev_time <- clock
        }
      }
      else
      {
        if ((n_down>0) & (n_up>0)) # people in both queues
        {
          if (elevator_state==0)
          {
            if (n_down>=elevator_capacity)
            {
```

```
        n_down <- n_down-elevator_capacity
      }
      else
      {
        n_down <- 0
      }
      elevator_state <- 0.5
      elev_time <- clock

    }
    else
    {
      if (elevator_state==1)
      {
        if (n_up>=elevator_capacity)
        {
          n_up <- n_up-elevator_capacity
        }
        else
        {
          n_up <- 0
        }
        elevator_state <- -0.5
        elev_time <- clock
      }
    }
  }
  if (n_down==0 & n_up>0) # people only upstairs
  {
    if (elevator_state==1)
    {
      if (n_up>=elevator_capacity)
      {
        n_up <- n_up-elevator_capacity
      }
      else
      {
        n_up <- 0
      }
      elevator_state <- -0.5
      elev_time <- clock
    }
    else
    {
      if (elevator_state==0)
      {
        elevator_state <- 0.5
        elev_time <- clock
      }
    }

  }
  if (n_down>0 & n_up==0) # people only downstairs
  {
          if (elevator_state==0)
          {
            if (n_down>=elevator_capacity)
            {
              n_down <- n_down-elevator_capacity
            }
            else
            {
              n_down <- 0
            }
            elevator_state <- 0.5
            elev_time <- clock

          }
          else
          {
            if (elevator_state==1)
            {
              elevator_state <- -0.5
              elev_time <- clock
            }
          }
        }
        time <-rexp(1,lambda_up+lambda_high*state1+ lambda_low*(1-state1)+
theta)
        if (clock + time <= elev_time+1 || elev_time==0)
        {
          clock <- clock+time
          spe_time <- 0
        }
        else
        {
          spe_time <- 1
          clock <- elev_time+1
          elevator_state <- elevator_state + 0.5
          elev_time <- 0
        }
        L_up <- L_up + n_up*(clock-last_clock)
        L_down <- L_down + n_down*(clock-last_clock)
        if (spe_time==0 && clock>0)
        {
          up_or_down <- runif(1,0,1)
          if (up_or_down>(avg_lambda_up/(avg_lambda_up+avg_lambda_down)))
          {
            n_down <- n_down + 1
          }
          else
          {
            n_up <- n_up + 1
          }
        }
      }
      #breaker <- breaker + 1
      last_clock <- clock
```

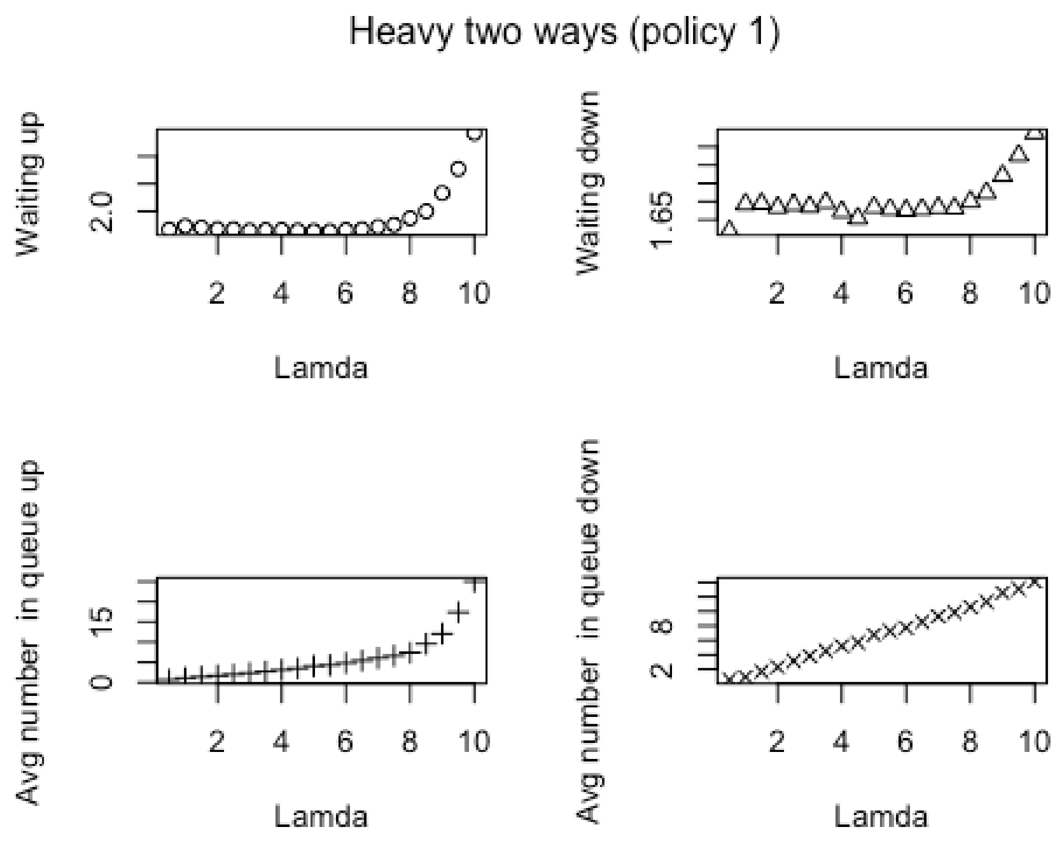
```r
    }
    W_up <- (L_up/T)/avg_lambda_up # average waiting time
    W_down <- (L_down/T)/avg_lambda_down
    res_up[i]= W_up
    res_down[i] <- W_down
    res_nup[i] <- n_up
    res_ndown[i] <- n_down
    counter <- 0
    n_up <- 0 # number of people up
    n_down <- 0 # number of people down
    l_train <- 0 # how much time passed since the last train
    elevator_state <- 0

    train_freq <- 30
    theta <- 1/10
    state1 <- 1

    elevator_travel <- 1
    elevator_capacity <- 16

    T <- 30

    clock <- 0

    L_up <- 0 # number of travelers in the system
    L_down <- 0
    elev_time <- 0
    spe_time <- 0
    last_clock <- 0


  }


  ret <- c(mean(res_up),mean(res_down),mean(res_nup),mean(res_ndown))
  return (ret)

}
c <-c(0.5,1,1.5,2,2.5,3,3.5,4,4.5,5,5.5,6,6.5,7,7.5,8,8.5,9,9.5,10)

matrixxx_1 <- matrix(1:80,ncol=4)
matrixxx_2 <- matrix(1:80,ncol=4)
matrixxx_3 <- matrix(1:80,ncol=4)
lamde_type<-1
for (i in c){
  l_high <- i*1
  l_low <- i*0.1
  l_up <- i*0.5
  a<-c((two_ways(l_high,l_low,l_up)))
  b<-c((up_peak(l_high,l_low,l_up)))
  c<-c((down_peak(l_high,l_low,l_up)))
  for(l in 1:4){
    matrixxx_1[lamde_type,l] <-a[l]
    matrixxx_2[lamde_type,l] <-b[l]
    matrixxx_3[lamde_type,l] <-c[l]
  }
  lamde_type<-lamde_type+1
}
matrix_graph<- function(matrixss,text)
{
X <- c(0.5,1,1.5,2,2.5,3,3.5,4,4.5,5,5.5,6,6.5,7,7.5,8,8.5,9,9.5,10)
Y1 <- matrixss[,1]
Y2 <- matrixss[,2]
Y3 <- matrixss[,3]
Y4 <- matrixss[,4]
par(mfrow=c(2,2))
plot(X,Y1, pch = 1,xlab = "Lamda",ylab = "Waiting up")
plot(X,Y2, pch = 2,xlab = "Lamda",ylab = "Waiting down")
plot(X,Y3, pch = 3,xlab = "Lamda",ylab ="Avg number  in queue up")
plot(X,Y4, pch = 4,xlab = "Lamda",ylab ="Avg number  in queue down")
mtext(text,side = 3,line = -2,outer = TRUE)
}
print("Raw data for the heavy two ways")
```

```
## [1] "Raw data for the heavy two ways"
```

```
print(matrixxx_1)
```

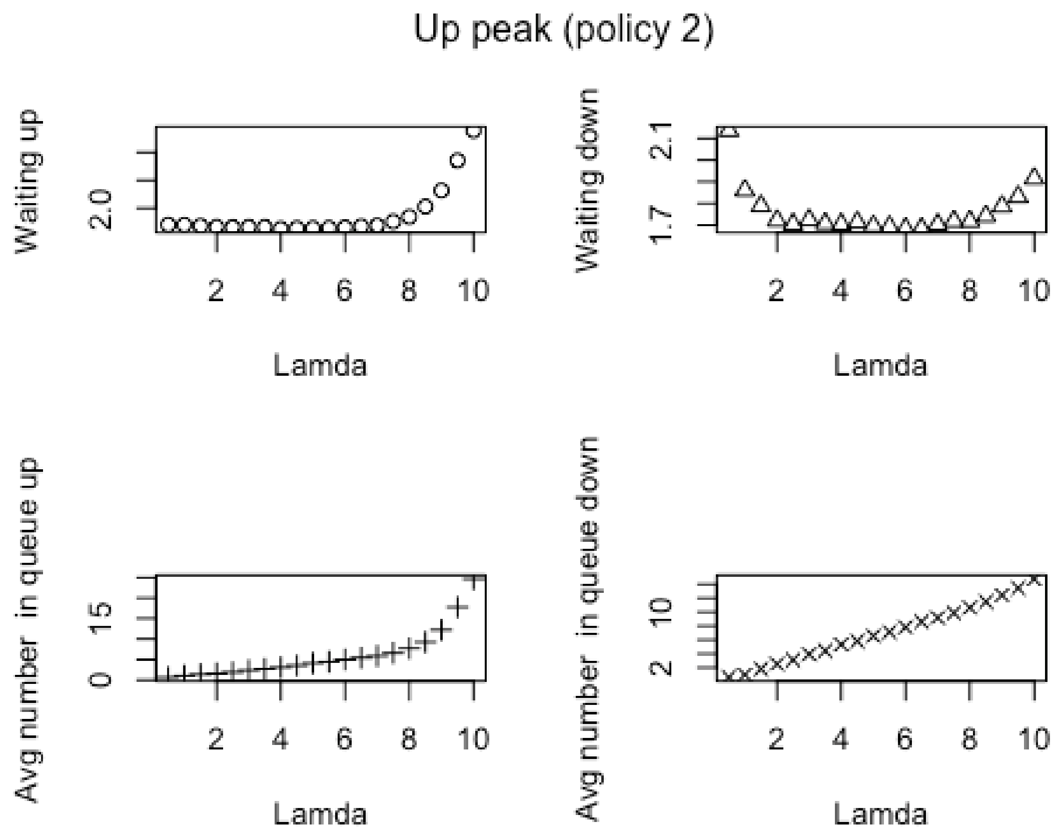
```
##           [,1]     [,2]   [,3]   [,4]
##  [1,] 1.661795 1.618680  0.688  0.554
##  [2,] 1.726686 1.692758  1.106  0.918
##  [3,] 1.699458 1.694414  1.376  1.672
##  [4,] 1.670841 1.683623  1.696  2.296
##  [5,] 1.669069 1.691253  1.950  3.144
##  [6,] 1.643303 1.686122  2.340  3.818
##  [7,] 1.653655 1.697374  2.686  4.528
##  [8,] 1.656084 1.671970  3.074  5.200
##  [9,] 1.645260 1.654168  3.488  5.684
## [10,] 1.637199 1.684253  4.082  6.754
## [11,] 1.633684 1.678940  4.278  7.256
## [12,] 1.655210 1.675624  4.792  7.730
## [13,] 1.668177 1.679258  5.400  8.546
## [14,] 1.714676 1.683129  6.034  9.320
## [15,] 1.751138 1.681711  6.564  9.864
## [16,] 1.871727 1.699863  7.384 10.604
## [17,] 1.998439 1.723978  9.572 11.340
## [18,] 2.330541 1.770819 12.068 12.564
## [19,] 2.771028 1.826213 17.396 13.110
## [20,] 3.423611 1.887409 24.998 14.128
```

```
matrix_graph(matrixxx_1,"Heavy two ways (policy 1)")
```

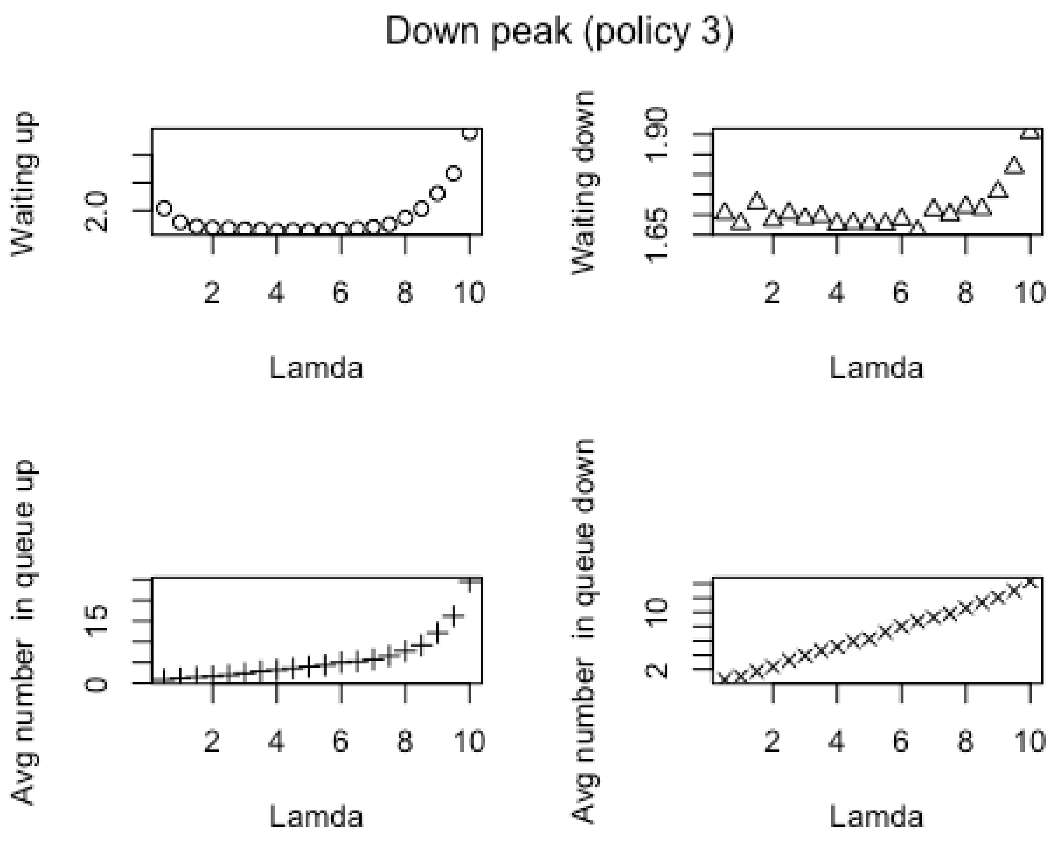


```
print("Raw data for the up peak")

## [1] "Raw data for the up peak"

print(matrixxx_2)

##            [,1]     [,2]   [,3]   [,4]
##  [1,] 1.701876 2.135611  0.678  0.644
##  [2,] 1.700174 1.860648  1.044  0.988
##  [3,] 1.682872 1.789018  1.362  1.816
##  [4,] 1.664474 1.721762  1.580  2.526
##  [5,] 1.658258 1.704428  1.870  3.110
##  [6,] 1.662427 1.724296  2.368  3.936
##  [7,] 1.661793 1.707531  2.688  4.430
##  [8,] 1.631042 1.704119  3.046  5.332
##  [9,] 1.640762 1.713301  3.526  5.826
## [10,] 1.641879 1.696161  4.120  6.564
## [11,] 1.645124 1.692593  4.448  7.126
## [12,] 1.653189 1.683732  4.968  7.810
## [13,] 1.674058 1.684594  5.412  8.612
## [14,] 1.690251 1.703284  5.856  9.178
## [15,] 1.758887 1.718567  6.572  9.872
## [16,] 1.844046 1.717794  7.714 10.628
## [17,] 2.027259 1.742057  9.262 11.480
## [18,] 2.316277 1.787278 12.204 12.424
## [19,] 2.852449 1.831068 17.788 13.442
## [20,] 3.387675 1.915371 24.634 14.744
```

```
matrix_graph(matrixxx_2,"Up peak (policy 2)")
```



```
print("Raw data for the down peak")

## [1] "Raw data for the down peak"

print(matrixxx_3)

##             [,1]     [,2]   [,3]   [,4]
##  [1,] 2.037842 1.703128  0.822  0.552
##  [2,] 1.789332 1.677833  1.110  0.982
##  [3,] 1.712857 1.729273  1.398  1.698
##  [4,] 1.686048 1.686143  1.602  2.350
##  [5,] 1.679867 1.703757  1.906  3.240
##  [6,] 1.660220 1.690779  2.344  3.904
##  [7,] 1.650512 1.695934  2.830  4.606
##  [8,] 1.632096 1.676218  3.122  5.156
##  [9,] 1.642039 1.678738  3.532  5.894
## [10,] 1.639641 1.678674  3.958  6.240
## [11,] 1.633601 1.676078  4.320  7.226
## [12,] 1.655224 1.688722  4.978  7.996
## [13,] 1.670572 1.659686  5.200  8.704
## [14,] 1.702586 1.712164  5.684  9.260
## [15,] 1.751077 1.700526  6.526  9.772
## [16,] 1.869759 1.719451  7.738 10.578
## [17,] 2.033432 1.714133  9.132 11.370
## [18,] 2.304941 1.757177 12.114 12.078
## [19,] 2.666851 1.817919 16.180 13.008
## [20,] 3.405832 1.904636 24.586 14.326

matrix_graph(matrixxx_3,"Down peak (policy 3)")
```



IV. Conclusion

As we saw, the lambda variable does not represent the absolute lambda, but a coefficient (c) which we multiplied all the lambda values:

Lambda_high = c*1

Lambda_low = c*0.1

Lambda_up = c*0.5

➔ When lambda_high and lamba_low represent the (non-constant) platform floor interarrival rates.

After running the simulation, we have reached some conclusions:

- There is no significant difference in the waiting times according to the different policies.
- When lambda values are low, a certain change can be seen:
    - In the first run (lamba coefficient = 0.5) the average waiting time at the entrance floor is slightly lower when the policy is “up peak” compared to “heavy two way”. When the police is “down peak”, the waiting time at the entrance floor is significantly larger. Such results last until about the 5th run (out of 20). When lambda values are high, it is difficult to see a significant difference.
- As we mentioned, the difference is mainly between the up-peak and down-peak policies. Surprisingly, it is the “heavy two way” policy that shows better waiting times. We can see that, in fact, the main effect of the up-peak/down-peak policies is increasing waiting times for the other “disadvantaged” queue, and doesn’t necessarily decrease waiting times in the preferred line.

Different explanations for the simulation results:

The reason for the difference specifically in the lower lambda values lies in the nature of the policy:

In the up-peak policy there is the difference that if both queues are empty and the elevator is at the platform floor, it will go up even if it has no passengers (and the opposite is true for the down-peak policy).

However, when lambda values are high, it is less likely that there will be empty queues, and so the policy that is chosen doesn’t really make a difference (elevator is moving non-stop).

Difference between simulation and theory:

In theory, it is possible that the waiting times are lower because of the assumption of batches. According to the M/D/1 queue model, each batch arrives on average like all 16 customers arrive in the simulation, but the waiting time of the entire batch is obviously lower than the average waiting time of all the customers that make up the batch, and this can be seen only in simulation. In addition to this, the theoretical model ignores the dependence between the queues, and perhaps this is the reason for the difference – as we mentioned, in the simulation (and also in reality) the queues waiting times are dependent of one another.