Assignment 3

ELL - 785 Computer Communication Networks

Saubhadra Gautam
2019JTM2167

Srijan Upadhyay
2019JTM2168

2019-2021

A report presented for the assignment on

Priority Queueing

Bharti School Of

Telecommunication Technology and Management

IIT Delhi

India

November 27, 2019

Contents

1 The M/M/1 Queueing system **4**

2 The M/G/1 Queueing system **4**

3 Referred Research paper **5**

3.1 Abstract 5

4 Flowchart **6**

5 Result **7**

6 Appendix **11**

6.1 Priority Queue code 11

6.2 M/M/1 Queue code 13

6.3 Simulation code 16

List of Figures

1 Flowchart 6
2 Service time per customer Output 7
3 Service end date per customer Output 7
4 Waiting time per customer Output 8
5 Service end date per customer Output 8
6 Output 9
7 Output 9
8 Comparison of waiting times of two priority 10

1 The M/M/1 Queueing system

The M/M/1 queueing system consists of a single queueing station with a single server. Customers arrive according to a poisson process with rate λ,and the probability distribution of the service time is exponential with mean $1/\mu$ sec.
The first M in the M/M/1 is for memoryless.
The second M indicates the nature of probability distribution of the service times is exponential.
The last number indicates the number of servers.
From little theorem, we have -

$$N = \lambda T$$
$$N_Q = \lambda W$$

where, N = Average number of customers in the system.
T = Average customer time in the system.
N_Q = Average number of customers waiting in queue.
W = Average customer waiting time in queue.
P_n = Probability of n customers in the system, n=0,1,...
From these probabilities, we can get

$$N = \sum_{n=0}^{\infty} nP_n$$

2 The M/G/1 Queueing system

This is a single server queueing system where customers arrive according to a poisson process with rate λ, but the customer service times have a general distribution - not necessarily exponential as in the M/M/1 system. Suppose that customers are served in the order they arrive and that X_i is the service time of the i^{th} arrival. We assume that the random variables $(X_1, X_2, ...)$ are identically distributed, mutually dependent, and independent of the interarrival times. Let

$$\bar{X} = E\{X\} = 1/\mu = \text{Average service time}$$
$$\bar{X^2} = E\{X^2\} = \text{Second moment of service time}$$

$$W = \lambda \bar{X^2}/2(1-\rho)$$

$$T = \bar{X} + (\lambda \bar{X^2}/2(1-\rho)$$

$$N_Q = \lambda^2 \bar{X^2}/2(1-\rho)$$

$$N = \rho + (\lambda^2 \bar{X^2}/2(1-\rho)$$

3 Referred Research paper

Topic - Performance Analysis of EDF Scheduling in a Multi-Priority Preemptive M/G/1 Queue

3.1 Abstract

his paper presents a queueing theoretic performance model for a multipriority preemptive M=G=1=:=EDFsystem. Existing models on EDF scheduling consider them to be M=M=1queues or nonpreemptive M=G=1queues. The proposed model approximates the mean waiting time for a given class based on the higher and lower priority tasks receiving service prior to the targetand the mean residual service time experienced. Additional time caused by preemptions is estimated as part of mean request completion time for a given class and as part of the mean delay experienced due to jobs in execution, on an arrival. The model is evaluated analytically and by simulation. Results confirm its accuracy, with the difference being a factor of two on average in high loads. Comparisons with other algorithms (such as First-Come-First-Served, Round-Robin and Nonpreemptive Priority Ordered)reveal that EDF achieves a better balance among priority classes where high priority requests are favored while preventing lowerpriority requests from overstarvation. EDF achieves best waiting times for higher priorities in lower to moderate loads (0.2-0.6)and while only being 6.5 times more than static priority algorithms in high loads (0.9). However, for the lowest priority classes, itachieves comparable waiting times to Round-Robin and First-Come-First-Served in low to moderate loads and achieves waitingtimes only twice the amount of Round-Robin in high system loads.

$$P_i = \lambda_i \left(\frac{\overline{X_i} + \sum_{j^*} (\rho_{j'} D_{i,j^*})}{(1 - \sum_{j'} \rho_{j'})} \right)$$
$$= \frac{\rho_i + \sum_{j^*} (\rho_{j'} D_{i,j^*}) \lambda_i}{(1 - \sum_{j'} \rho_{j'})}.$$

$$\overline{W_0}^i = \sum_{k=1}^{i} (P_k \overline{R_k}).$$

$$\overline{W_0}^i = \sum_{k=1}^{i} \overline{R_k} \left(\frac{\rho_i + \sum_{j^*} (\rho_{j'} D_{i,j^*}) \lambda_i}{(1 - \sum_{j'} \rho_{j'})} \right)$$

4 Flowchart

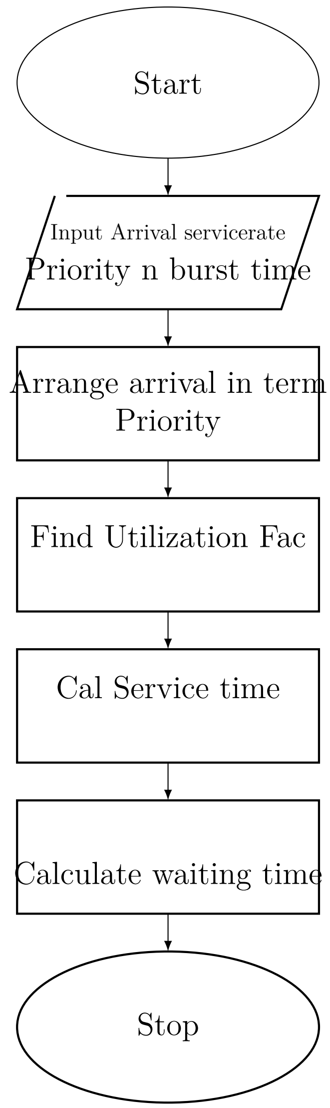

Figure 1: Flowchart

5 Result

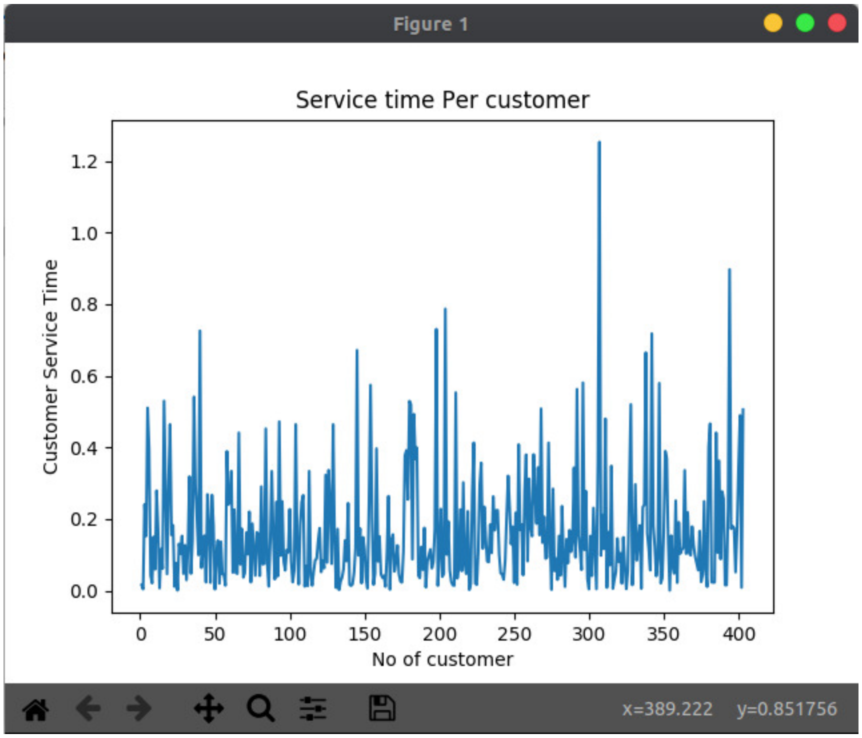

Figure 2: Service time per customer Output

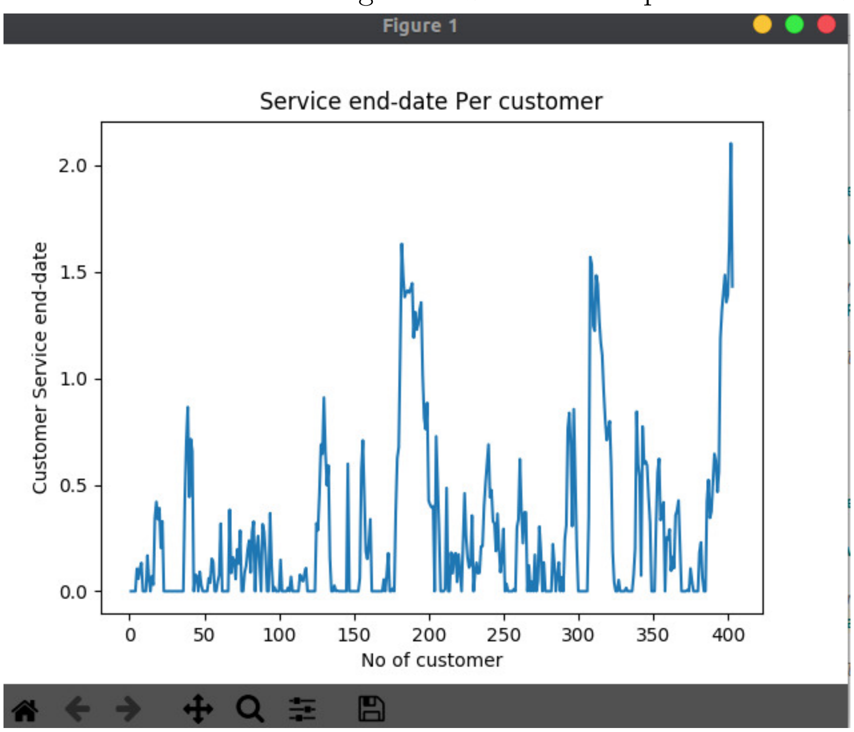

Figure 3: Service end date per customer Output

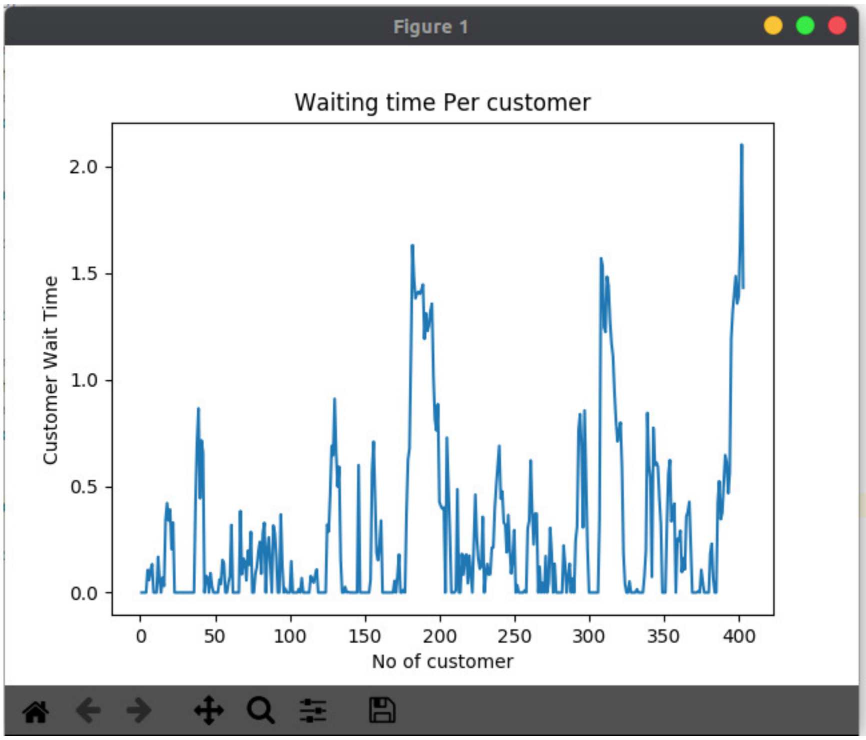

Figure 4: Waiting time per customer Output

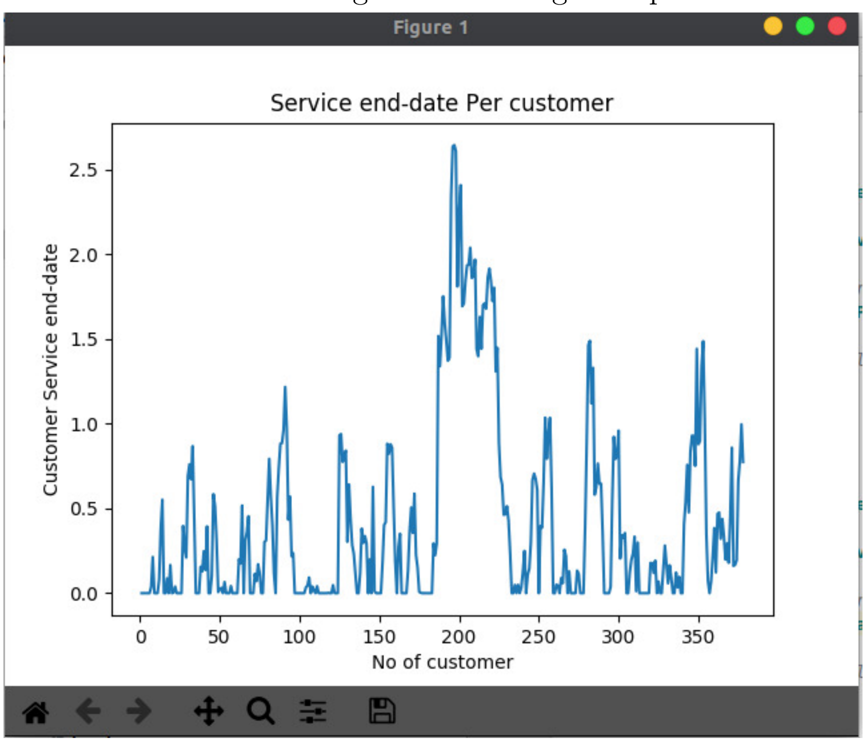

Figure 5: Service end date per customer Output

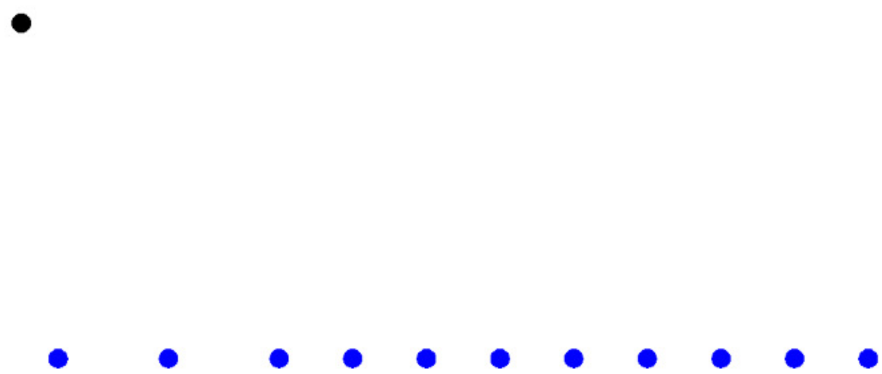

Figure 6: Output

```
Enter the number of processess:
5

Enter the burst time of the processes:

10 12 8 7 15

Enter the priority of the processes:

1 5 4 2 3


Process   Burst Time      Waiting Time    Turn Around Time
1       10      0       10


4       7       10      17


5       15      17      32


3       8       32      40


2       12      40      52


Average Waiting time is: 30.2
Average Turn Around Time is: 151

Process finished with exit code 0
```

Figure 7: Output

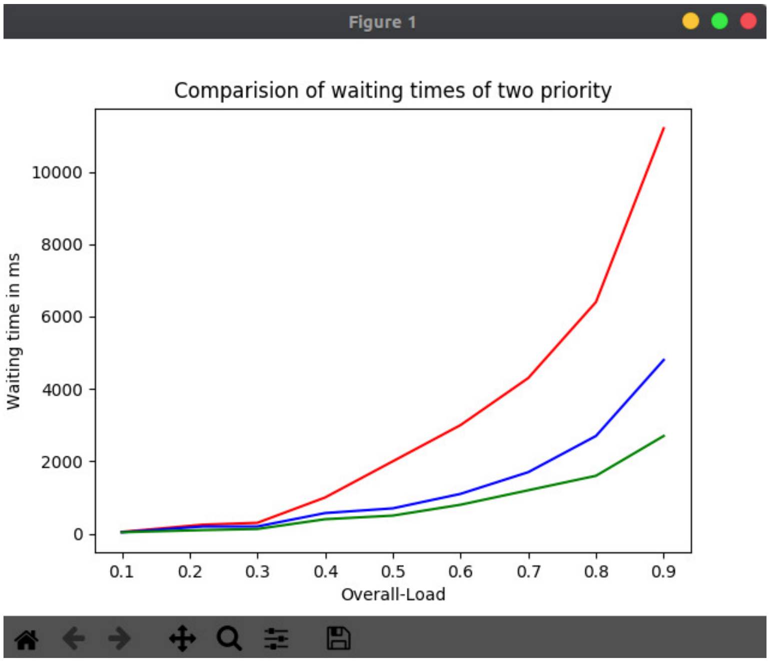

Figure 8: Comparison of waiting times of two priority

6 Appendix

6.1 Priority Queue code

```

# bt,wt,tat stands for burst time, waiting time, turn around time
    respectively
import matplotlib.pyplot as plt

import random
import matplotlib.pyplot as plt

# x axis values
x = [0.1, 0.22, 0.3,0.4,0.5,0.6,0.7,0.8,0.9]
# corresponding y axis values
y = [50,250 ,300,1000,2000,3000,4300,6400,11200]
z = [40,200 ,200,570,700,1100,1700,2700,4800]
k = [40,100 ,130,400,500,800,1200,1600,2700]
def neg_exp(lambd):
    return random.expovariate(lambd)


class Customer:
    def __init__(self,arrival_date,service_start_date,service_time):
        self.arrival_date = arrival_date
        self.service_start_date = service_start_date
        self.service_time = service_time
        self.service_end_date = self.service_start_date + self.service_time
        self.wait = self.service_start_date - self.arrival_date
x

print("Enter the number of processess: ")
n = int(input())
processes = []
for i in range(0, n):
    processes.insert(i, i + 1)

# Input Burst time of every process
print("\nEnter the burst time of the processes: \n")
bt = list(map(int, input().split()))

# Input Priority of every process
print("\nEnter the priority of the processes: \n")
priority = list(map(int, input().split()))
tat = []
wt = []

print("Enter arrival rate")
lambd = list(map(int, input().split()))

print("Enter service rate")

mu = list(map(int, input().split()))

simulation_time = input("Enter Simulation time")
```

```


# Sorting processes burst time, on the basis of their priority
for i in range(0, len(priority) - 1):
    for j in range(0, len(priority) - i - 1):
        if (priority[j] > priority[j + 1]):
            swap = priority[j]
            priority[j] = priority[j + 1]
            priority[j + 1] = swap

            swap = bt[j]
            bt[j] = bt[j + 1]
            bt[j + 1] = swap

            swap = processes[j]
            processes[j] = processes[j + 1]
            processes[j + 1] = swap

wt.insert(0, 0)
tat.insert(0, bt[0])

# Calculating of waiting time and Turn Around Time of each process
for i in range(1, len(processes)):
    wt.insert(i, wt[i - 1] + bt[i - 1])
    tat.insert(i, wt[i] + bt[i])

# calculating average waiting time and average turn around time
avgtat = 0
avgwt = 0
for i in range(0, len(processes)):
    avgwt = avgwt + wt[i]
    avgtat = avgtat + tat[i]
avgwt = float(avgwt) / n
avgwt = float(avgtat) / n
print("\n")
print("Process\t  Burst Time\t  Waiting Time\t  Turn Around Time")
for i in range(0, n):
    print(str(processes[i]) + "\t\t" + str(bt[i]) + "\t\t" + str(wt[i]) + "\t\
    t" + str(tat[i]))
    print("\n")
print("Average Waiting time is: " + str(avgwt))
print("Average Turn Around Time is: " + str(avgtat))

#
# t = 0
# customers= []
# sim= int(simulation_time)
# rho = []
# while t < sim:
#     # print("Simulation start5")
#     for i in range(0, n):
#         if len(customers) == 0:
#             arrival_date = neg_exp(lambd)
#             service_start_date = arrival_date
#
#         else:
```

```
#             arrival_date += neg_exp(lambd)
#             # print(arrival_date)
#             service_start_date = max(arrival_date, customers[-1].service_end_date)
#         service_time = (mu*2)
#
#         # create new customer
#         customers.append(Customer(arrival_date, service_start_date, service_time))
#
#         # increment clock till next end of service
#         t = arrival_date
#
#         # calculation of rho
#         rho[i] =  lambd[i]/mu[i]

    # plotting the points
plt.plot(x, y, 'r') # plotting t, a separately
plt.plot(x, z, 'b') # plotting t, b separately
plt.plot(x, k, 'g') # plotting t, c separately

    # naming the x axis
plt.ylabel('Waiting time in ms')

    # naming the y axis
plt.xlabel('Overall-Load')

    # giving a title to my graph
plt.title('Comparision of waiting times of two priority')

    # function to show the plot
plt.show()



# < b > < / b >
```

6.2 M/M/1 Queue code

```
import matplotlib.pyplot as plt

import random
import csv
# importing the required module


# define a class called 'Customer'

x = []
y = []
z = []
k = []
l = []

class Customer:
```

```
    def __init__(self,arrival_date,service_start_date,service_time):
        self.arrival_date = arrival_date
        self.service_start_date = service_start_date
        self.service_time = service_time
        self.service_end_date = self.service_start_date + self.service_time
        self.wait = self.service_start_date - self.arrival_date

# a simple function to sample from negative exponential


def neg_exp(lambd):
    return random.expovariate(lambd)

def QSim(lambd=False,mu=False,simulation_time=False):

    print("Simulation start")
    if not lambd:
        lambd= int(input("Enter arrival rate"))

    if not mu:
        mu = int(input("Enter arrival rate"))

    if not simulation_time:
        simulation_time = input("Enter arrival rate")
    # print(lambd)
    # print(simulation_time)
    # print("Simulation start4")
    t = 0
    customers= []
    sim= int(simulation_time)
    # while(t < simulation_time):
    while t < sim:
        # print("Simulation start5")
        if len(customers) == 0:
            arrival_date = neg_exp(lambd)
            service_start_date = arrival_date

        else:
            arrival_date+= neg_exp(lambd)
            service_start_date = max(arrival_date, customers[-1].service_end_date)
        service_time = neg_exp(mu)

        # create new customer
        customers.append(Customer(arrival_date, service_start_date, service_time))
        print(customers)
        # increment clock till next end of service
        t = arrival_date

        # calculate summary statistics
    Waits = [a.wait for a in customers]
    print(Waits)
    Mean_Wait = sum(Waits) / len(Waits)

    Total_Times = [a.wait + a.service_time for a in customers]
```

```
    Mean_Time = sum(Total_Times) / len(Total_Times)

    Service_Times = [a.service_time for a in customers]
    Mean_Service_Time = sum(Service_Times) / len(Service_Times)

    Utilisation = sum(Service_Times) / t

    # output summary statistics to screen
    # print("Simulation start3")
    print("")
    print("Summary results")
    print("")
    print("No of customer",len(customers))
    print("Mean Services",Mean_Service_Time)
    print("Mean Wait: ", Mean_Wait)
    print("Mean Time in System: ",Mean_Time)
    print("Utilisation: ",Utilisation)
    print("")

    # prompt user to output full data set to csv
    if input("Output data to csv (True/False)? "):
        outfile = open('MM1Q-output-(%s,%s,%s).csv' % (lambd, mu,
    simulation_time), 'w')
        output = csv.writer(outfile)
        output.writerow(
            ['Customer', 'Arrival_Date', 'Wait', 'Service_Start_Date', '
    Service_Time', 'Service_End_Date'])
        i = 0

        for customer in customers:
            i = i + 1
            outrow = []
            y.append(customer.wait)
            x.append(i)
            z.append(customer.service_time)
            k.append(customer.service_end_date)


            outrow.append(i)
            outrow.append(customer.arrival_date)
            outrow.append(customer.wait)
            outrow.append(customer.service_start_date)
            outrow.append(customer.service_time)
            outrow.append(customer.service_end_date)
            output.writerow(outrow)
        outfile.close()

        # Function to plot
        plt.plot(x, y)
        # naming the x axis
        plt.xlabel('No of customer')
        # naming the y axis
        plt.ylabel('Customer Wait Time')
        # giving a title to my graph
        plt.title('Waiting time Per customer')
        # function to show the plot
```

```
        plt.show()

        # Function to plot
        plt.plot(x, z)
        # naming the x axis
        plt.xlabel('No of customer')
        # naming the y axis
        plt.ylabel('Customer Service Time')

        # giving a title to my graph
        plt.title('Service time Per customer')

        # function to show the plot
        plt.show()

        # Function to plot
        plt.plot(x, y)
        # naming the x axis
        plt.xlabel('No of customer')
        # naming the y axis
        plt.ylabel('Customer Service end-date')

        # giving a title to my graph
        plt.title('Service end-date Per customer')

        # function to show the plot
        plt.show()

    print("")

    return

QSim()
```

6.3 Simulation code

```
#!/usr/bin/env python
"""
Library with some objects that make use of the python Turtle library to show graphics of a discrete event simulation of an MM1 queue (random arrivals and services, a single server).
There are various objects that allow for the simulation and demonstration of emergent behaviour:
- Player (I use the term player instead of customer as I also allow for the selfish and optimal behaviour: graphical representation: blue coloured dot);
- SelfishPlayer (inherited from Player: when passed a value of service, a SelfishPlayer will join the queue if and only if it is in their selfish interest: graphical representation: red coloured dot.);
- OptimalPlayer (uses a result from Naor to ensure that the mean cost is reduced: graphical representation: gold coloured dot.)
- Queue
- Server
- Sim (this is the main object that generates all other objects as required).
"""
```

```python
from __future__ import division  # Simplify division
from turtle import Turtle, mainloop, setworldcoordinates  # Commands needed from Turtle
from random import expovariate as randexp, random  # Pseudo random number generation
import sys  # Use to write to out

def mean(lst):
    """
    Function to return the mean of a list.
    Argument: lst - a list of numeric variables
    Output: the mean of lst
    """
    if len(lst) > 0:
        return sum(lst) / len(lst)
    return False

def movingaverage(lst):
    """
    Custom built function to obtain moving average
    Argument: lst - a list of numeric variables
    Output: a list of moving averages
    """
    return [mean(lst[:k]) for k in range(1 , len(lst) + 1)]

def plotwithnobalkers(queuelengths, systemstates, timepoints, savefig, string):
    """
    A function to plot histograms and timeseries.
    Arguments:
        - queuelengths (list of integers)
        - systemstates (list of integers)
        - timtepoints (list of integers)
    """
    try:
        import matplotlib.pyplot as plt
    except:
        sys.stdout.write("matplotlib does not seem to be installed: no  plots can be produced.")
        return

    plt.figure(1)
    plt.subplot(221)
    plt.hist(queuelengths, normed=True, bins=min(20, max(queuelengths)))
    plt.title("Queue length")
    plt.subplot(222)
    plt.hist(systemstates, normed=True, bins=min(20, max(systemstates)))
    plt.title("System state")
    plt.subplot(223)
    plt.plot(timepoints, movingaverage(queuelengths))
    plt.title("Mean queue length")
    plt.subplot(224)
    plt.plot(timepoints, movingaverage(systemstates))
    plt.title("Mean system state")
    if savefig:
        plt.savefig(string)
```

```
    else:
        plt.show()

def plotwithbalkers(selfishqueuelengths, optimalqueuelengths, selfishsystemstates, optimalsystemstates, timepoints, savefig, string):
    """
    A function to plot histograms and timeseries when you have two types of players
    Arguments:
        - selfishqueuelengths (list of integers)
        - optimalqueuelengths (list of integers)
        - selfishsystemstates (list of integers)
        - optimalsystemstates (list of integers)
        - timtepoints (list of integers)
        - savefig (boolean)
        - string (a string)
    """
    try:
        import matplotlib.pyplot as plt
    except:
        sys.stdout.write("matplotlib does not seem to be installed: no  plots can be produced.")
        return
    queuelengths = [sum(k) for k in zip(selfishqueuelengths, optimalqueuelengths)]
    systemstates = [sum(k) for k in zip(selfishsystemstates, optimalsystemstates)]
    fig = plt.figure(1)
    plt.subplot(221)
    plt.hist([selfishqueuelengths, optimalqueuelengths, queuelengths], normed=True, bins=min(20, max(queuelengths)), label=['Selfish players','Optimal players','Total players'], color=['red', 'green', 'blue'])
   #plt.legend()
    plt.title("Number in queue")
    plt.subplot(222)
    plt.hist([selfishsystemstates, optimalsystemstates, systemstates], normed=True, bins=min(20, max(systemstates)), label=['Selfish players','Optimal players','Total players'], color=['red', 'green', 'blue'])
   #plt.legend()
    plt.title("Number in system")
    plt.subplot(223)
    plt.plot(timepoints, movingaverage(selfishqueuelengths), label='Selfish players', color='red')
    plt.plot(timepoints, movingaverage(optimalqueuelengths), label='Optimal players', color='green')
    plt.plot(timepoints, movingaverage(queuelengths), label='Total', color='blue')
   #plt.legend()
    plt.title("Mean number in queue")
    plt.subplot(224)
    line1, =  plt.plot(timepoints, movingaverage(selfishsystemstates), label='Selfish players', color='red')
    line2, = plt.plot(timepoints, movingaverage(optimalsystemstates), label='Optimal players', color='green')
    line3, = plt.plot(timepoints, movingaverage(systemstates), label='Total', color='blue')
```

```
    #plt.legend()
    plt.title("Mean number in system")
    fig.legend([line1,line2,line3],['Selfish players', 'Optimal players', '
    Total'],loc='lower center',fancybox=True,ncol=3, bbox_to_anchor=(.5,0))
    plt.subplots_adjust(bottom=.15)
    if savefig:
        plt.savefig(string)
    else:
        plt.show()

def naorthreshold(lmbda, mu, costofbalking):
    """
    Function to return Naor's threshold for optimal behaviour in an M/M/1
    queue. This is taken from Naor's 1969 paper: 'The regulation of queue size
    by Levying Tolls'
    Arguments:
        lmbda - arrival rate (float)
        mu - service rate (float)
        costofbalking - the value of service, converted to time units. (float)
    Output: A threshold at which optimal customers must no longer join the
    queue (integer)
    """
    n = 0  # Initialise n
    center = mu * costofbalking  # Center mid point of inequality from Naor's
    aper
    rho = lmbda / mu
    while True:
        LHS = (n*(1-rho)- rho * (1-rho**n))/((1-rho)**2)
        RHS = ((n+1)*(1- rho)-rho*(1-rho**(n+1)))/((1-rho)**2)
        if LHS <= center and center <RHS:
            return n
        n += 1  # Continually increase n until LHS and RHS are either side of
    center


class Player(Turtle):
    """
    A generic class for our 'customers'. I refer to them as players as I like
    to consider queues in a game theoretical framework. This class is inherited
     from the Turtle class so as to have the graphical interface.
    Attributes:
        lmbda: arrival rate (float)
        mu: service rate (float)
        queue: a queue object
        server: a server object
    Methods:
        move - move player to a given location
        arrive - a method to make our player arrive at the queue
        startservice - a method to move our player from the queue to the
    server
        endservice - a method to complete service
    """
    def __init__(self, lmbda, mu, queue, server, speed):
        """
        Arguments:
            lmbda: arrival rate (float)
```

```
            interarrivaltime: a randomly sampled interarrival time (negative exponential for now)
            mu: service rate (float)
            service: a randomly sampled service time (negative exponential for now)
            queue: a queue object
            shape: the shape of our turtle in the graphics (a circle)
            server: a server object
            served: a boolean that indicates whether or not this player has been served.
            speed: a speed (integer from 0 to 10) to modify the speed of the graphics
            balked: a boolean indicating whether or not this player has balked (not actually needed for the base Player class... maybe remove... but might be nice to keep here...)
        """
        Turtle.__init__(self)  # Initialise all base Turtle attributes
        self.interarrivaltime = randexp(lmbda)
        self.lmbda = lmbda
        self.mu = mu
        self.queue = queue
        self.served = False
        self.server = server
        self.servicetime = randexp(mu)
        self.shape('circle')
        self.speed(speed)
        self.balked = False

    def move(self, x, y):
        """
            A method that moves our player to a given point
            Arguments:
                x: the x position on the canvas to move the player to
                y: the y position on the canvas to move the player to.
            Output: NA
        """
        self.setx(x)
        self.sety(y)

    def arrive(self, t):
        """
        A method that make our player arrive (the player is first created to generate an interarrival time, service time etc...).
        Arguments: t the time of arrival (a float)
        Output: NA
        """
        self.penup()
        self.arrivaldate = t
        self.move(self.queue.position[0] + 5, self.queue.position[1])
        self.color('blue')
        self.queue.join(self)

    def startservice(self, t):
        """
        A method that makes our player start service (This moves the graphical representation of the player and also make the queue update it's graphics)
```

```
    .
        Arguments: t the time of service start (a float)
        Output: NA
        """
        if not self.served and not self.balked:
            self.move(self.server.position[0], self.server.position[1])
            self.servicedate = t + self.servicetime
            self.server.start(self)
            self.color('gold')
            self.endqueuedate = t

    def endservice(self):
        """
        A method that makes our player end service (This moves the graphical representation of the player and updates the server to be free).
        Arguments: NA
        Output: NA
        """
        self.color('grey')
        self.move(self.server.position[0] + 50 + random(), self.server.position[1] - 50 + random())
        self.server.players = self.server.players[1:]
        self.endservicedate = self.endqueuedate + self.servicetime
        self.waitingtime = self.endqueuedate - self.arrivaldate
        self.served = True

class SelfishPlayer(Player):
    """
    A class for a player who acts selfishly (estimating the amount of time that they will wait and comparing to a value of service). The only modification is the arrive method that now allows players to balk.
    """
    def __init__(self, lmbda, mu, queue, server, speed, costofbalking):
        Player.__init__(self, lmbda, mu, queue, server, speed)
        self.costofbalking = costofbalking
    def arrive(self, t):
        """
        As described above, this method allows players to balk if the expected time through service is larger than some alternative.
        Arguments: t - time of arrival (a float)
        Output: NA
        """
        self.penup()
        self.arrivaldate = t
        self.color('red')
        systemstate = len(self.queue) + len(self.server)
        if (systemstate + 1) / (self.mu) < self.costofbalking:
            self.queue.join(self)
            self.move(self.queue.position[0] + 5, self.queue.position[1])
        else:
            self.balk()
            self.balked = True
    def balk(self):
        """
        Method to make player balk.
        Arguments: NA
```

```
        Outputs: NA
        """
        self.move(random(), self.queue.position[1] - 25 + random())

class OptimalPlayer(Player):
    """
    A class for a player who acts within a socially optimal framework (using the threshold from Naor's paper). The only modification is the arrive method that now allows players to balk and a new attribute for the Naor threshold.
    """
    def __init__(self, lmbda, mu, queue, server, speed, naorthreshold):
        Player.__init__(self, lmbda, mu, queue, server, speed)
        self.naorthreshold = naorthreshold
    def arrive(self, t):
        """
        A method to make player arrive. If more than Naor threshold are present in queue then the player will balk.
        Arguments: t - time of arrival (float)
        Outputs: NA
        """
        self.penup()
        self.arrivaldate = t
        self.color('green')
        systemstate = len(self.queue) + len(self.server)
        if systemstate < self.naorthreshold:
            self.queue.join(self)
            self.move(self.queue.position[0] + 5, self.queue.position[1])
        else:
            self.balk()
            self.balked = True
    def balk(self):
        """
        A method to make player balk.
        """
        self.move(10 + random(), self.queue.position[1] - 25 + random())

class Queue():
    """
    A class for a queue.
    Attributes:
        players - a list of players in the queue
        position - graphical position of queue
    Methods:
        pop - returns first in player from queue and updates queue graphics
        join - makes a player join the queue
    """
    def __init__(self, qposition):
        self.players = []
        self.position = qposition
    def __iter__(self):
        return iter(self.players)
    def __len__(self):
        return len(self.players)
    def pop(self, index):
        """
```

```
        A function to return a player from the queue and update graphics.
        Arguments: index - the location of the player in the queue
        Outputs: returns the relevant player
        """
        for p in self.players[:index] + self.players[index + 1:]:  # Shift everyone up one queue spot
            x = p.position()[0]
            y = p.position()[1]
            p.move(x + 10, y)
        self.position[0] += 10  # Reset queue position for next arrivals
        return self.players.pop(index)
    def join(self, player):
        """
        A method to make a player join the queue.
        Arguments: player object
        Outputs: NA
        """
        self.players.append(player)
        self.position[0] -= 10

class Server():
    """
    A class for the server (this could theoretically be modified to allow for more complex queues than M/M/1)
    Attributes:
        - players: list of players in service (at present will be just the one player)
        - position: graphical position of queue
    Methods:
        - start: starts the service of a given player
        - free: a method that returns free if the server is free
    """
    def __init__(self, svrposition):
        self.players = []
        self.position = svrposition
    def __iter__(self):
        return iter(self.players)
    def __len__(self):
        return len(self.players)
    def start(self,player):
        """
        A function that starts the service of a player (there is some functionality already in place in case multi server queue ever gets programmed). Moves all graphical stuff.
        Arguments: A player object
        Outputs: NA
        """
        self.players.append(player)
        self.players = sorted(self.players, key = lambda x : x.servicedate)
        self.nextservicedate =  self.players[0].servicedate
    def free(self):
        """
        Returns True if server is empty.
        """
        return len(self.players) == 0

```

```python
class Sim():
    """
    The main class for a simulation.
    Attributes:
        - costofbalking (by default set to False for a basic simulation). Can be a float (indicating the cost of balking) in which case all players act selfishly. Can also be a list: l. In which case l[0] represents proportion of selfish players (other players being social players). l[1] then indicates cost of balking.
        - naorthresholed (by default set to False for a basic simulation). Can be an integer (not to be input but calculated using costofbalking).
        - T total run time (float)
        - lmbda: arrival rate (float)
        - mu: service rate (float)
        - players: list of players (list)
        - queue: a queue object
        - queuelengthdict: a dictionary that keeps track of queue length at given times (for data handling)
        - systemstatedict: a dictionary that keeps track of system state at given times (for data handling)
        - server: a server object
        - speed: the speed of the graphical animation
    Methods:
        - run: runs the simulation model
        - newplayer: generates a new player (that does not arrive until the clock advances past their arrivaldate)
        - printprogress: print the progress of the simulation to stdout
        - collectdata: collects data at time t
        - plot: plots summary graphs
    """

    def __init__(self, T, lmbda, mu, speed=6, costofbalking=False):
        ##################
        bLx = -10 # This sets the size of the canvas (I think that messing with this could increase speed of turtles)
        bLy = -110
        tRx = 230
        tRy = 5
        setworldcoordinates(bLx,bLy,tRx,tRy)
        qposition = [(tRx+bLx)/2, (tRy+bLy)/2]  # The position of the queue
        ##################
        self.costofbalking = costofbalking
        self.T = T
        self.completed = []
        self.balked = []
        self.lmbda = lmbda
        self.mu = mu
        self.players = []
        self.queue = Queue(qposition)
        self.queuelengthdict = {}
        self.server = Server([qposition[0] + 50, qposition[1]])
        self.speed = max(0,min(10,speed))
        self.naorthreshold = False
        if type(costofbalking) is list:
            self.naorthreshold = naorthreshold(lmbda, mu, costofbalking[1])
        else:
```

```
            self.naorthreshold = naorthreshold(lmbda, mu, costofbalking)
        self.systemstatedict = {}

    def newplayer(self):
        """
        A method to generate a new player (takes in to account cost of balking). So if no cost of balking is passed: only generates a basic player. If a float is passed as cost of balking: generates selfish players with that float as worth of service. If a list is passed then it creates a player (either selfish or optimal) according to a random selection.
        Arguments: NA
        Outputs: NA
        """
        if len(self.players) == 0:
            if not self.costofbalking:
                self.players.append(Player(self.lmbda, self.mu, self.queue, self.server, self.speed))
            elif type(self.costofbalking) is list:
                if random() < self.costofbalking[0]:
                    self.players.append(SelfishPlayer(self.lmbda, self.mu, self.queue, self.server, self.speed, self.costofbalking[1]))
                else:
                    self.players.append(OptimalPlayer(self.lmbda, self.mu, self.queue, self.server, self.speed, self.naorthreshold))
            else:
                self.players.append(SelfishPlayer(self.lmbda, self.mu, self.queue, self.server, self.speed, self.costofbalking))

    def printprogress(self, t):
        """
        A method to print to screen the progress of the simulation.
        Arguments: t (float)
        Outputs: NA
        """
        sys.stdout.write('\r%.2f%% of simulation completed (t=%s of %s)' % (100 * t/self.T, t, self.T))
        sys.stdout.flush()

    def run(self):
        """
        The main method which runs the simulation. This will collect relevant data throughout the simulation so that if matplotlib is installed plots of results can be accessed. Furthermore all completed players can be accessed in self.completed.
        Arguments: NA
        Outputs: NA
        """
        t = 0
        self.newplayer()  # Create a new player
        nextplayer = self.players.pop()  # Set this player to be the next player
        nextplayer.arrive(t)  # Make the next player arrive for service (potentially at the queue)
        nextplayer.startservice(t)  # This player starts service immediately
        self.newplayer()  # Create a new player that is now waiting to arrive
        while t < self.T:
```

```
            t += 1
            self.printprogress(t)  # Output progress to screen
            # Check if service finishes
            if not self.server.free() and t > self.server.nextservicedate:
                self.completed.append(self.server.players[0]) # Add completed player to completed list
                self.server.players[0].endservice()  # End service of a player in service
                if len(self.queue)>0:  # Check if there is a queue
                    nextservice = self.queue.pop(0)  # This returns player to go to service and updates queue.
                    nextservice.startservice(t)
                    self.newplayer()
            # Check if player that is waiting arrives
            if t > self.players[-1].interarrivaltime + nextplayer.arrivaldate:
                nextplayer = self.players.pop()
                nextplayer.arrive(t)
                if nextplayer.balked:
                    self.balked.append(nextplayer)
                if self.server.free():
                    if len(self.queue) == 0:
                        nextplayer.startservice(t)
                    else:  # Check if there is a queue
                        nextservice = self.queue.pop(0)  # This returns player to go to service and updates queue.
                        nextservice.startservice(t)
            self.newplayer()
            self.collectdata(t)

    def collectdata(self,t):
        """
        Collect data at each time step: updates data dictionaries.
        Arguments: t (float)
        Outputs: NA
        """
        if self.costofbalking:
            selfishqueuelength = len([k for k in self.queue if type(k) is SelfishPlayer])
            self.queuelengthdict[t] = [selfishqueuelength,len(self.queue) - selfishqueuelength]
            if self.server.free():
                self.systemstatedict[t] = [0,0]
            else:
                self.systemstatedict[t] = [self.queuelengthdict[t][0] + len([p for p in self.server.players if type(p) is SelfishPlayer]),self.queuelengthdict[t][1] + len([p for p in self.server.players if type(p) is OptimalPlayer])]
        else:
            self.queuelengthdict[t] = len(self.queue)
            if self.server.free():
                self.systemstatedict[t] = 0
            else:
                self.systemstatedict[t] = self.queuelengthdict[t] + 1

    def plot(self, savefig, warmup=0):
        """
```

```
        Plot the data
        """
        string = "lmbda=%s-mu=%s-T=%s-cost=%s.pdf" % (self.lmbda, self.mu, self.T, self.costofbalking) # An identifier
        if self.costofbalking:
            selfishqueuelengths = []
            optimalqueuelengths = []
            selfishsystemstates = []
            optimalsystemstates = []
            timepoints = []
            for t in self.queuelengthdict:
                if t >= warmup:
                    selfishqueuelengths.append(self.queuelengthdict[t][0])
                    optimalqueuelengths.append(self.queuelengthdict[t][1])
                    selfishsystemstates.append(self.systemstatedict[t][0])
                    optimalsystemstates.append(self.systemstatedict[t][1])
                    timepoints.append(t)
            plotwithbalkers(selfishqueuelengths, optimalqueuelengths, selfishsystemstates, optimalsystemstates, timepoints, savefig, string)
        else:
            queuelengths = []
            systemstates = []
            timepoints = []
            for t in self.queuelengthdict:
                if t >= warmup:
                    queuelengths.append(self.queuelengthdict[t])
                    systemstates.append(self.systemstatedict[t])
                    timepoints.append(t)
            plotwithnobalkers(queuelengths, systemstates, timepoints, savefig, string)

    def printsummary(self, warmup=0):
        """
        A method to print summary statistics.
        """
        if not self.costofbalking:
            self.queuelengths = []
            self.systemstates = []
            for t in self.queuelengthdict:
                if t >= warmup:
                    self.queuelengths.append(self.queuelengthdict[t])
                    self.systemstates.append(self.systemstatedict[t])
            self.meanqueuelength = mean(self.queuelengths)
            self.meansystemstate = mean(self.systemstates)
            self.waitingtimes = []
            self.servicetimes = []
            for p in self.completed:
                if p.arrivaldate >= warmup:
                    self.waitingtimes.append(p.waitingtime)
                    self.servicetimes.append(p.servicetime)
            self.meanwaitingtime = mean(self.waitingtimes)
            self.meansystemtime = mean(self.servicetimes) + self.meanwaitingtime
            sys.stdout.write("\n%sSummary statistics%s\n" % (10*"-",10*"-"))
            sys.stdout.write("Mean queue length: %.02f\n" % self.meanqueuelength)
```

```
            sys.stdout.write("Mean system state: %.02f\n" % self.meansystemstate)
            sys.stdout.write("Mean waiting time: %.02f\n" % self.meanwaitingtime)
            sys.stdout.write("Mean system time: %.02f\n" % self.meansystemtime)
            sys.stdout.write(39 * "-" + "\n")
        else:

            self.selfishqueuelengths = []
            self.optimalqueuelengths = []
            self.selfishsystemstates = []
            self.optimalsystemstates = []
            for t in self.queuelengthdict:
                if t >= warmup:
                    self.selfishqueuelengths.append(self.queuelengthdict[t][0])
                    self.optimalqueuelengths.append(self.queuelengthdict[t][1])
                    self.selfishsystemstates.append(self.systemstatedict[t][0])
                    self.optimalsystemstates.append(self.systemstatedict[t][1])
            self.meanselfishqueuelength = mean(self.selfishqueuelengths)
            self.meanoptimalqueuelength = mean(self.optimalqueuelengths)
            self.meanqueuelength = mean([sum(k) for k in zip(self.selfishqueuelengths, self.optimalqueuelengths)])
            self.meanselfishsystemstate = mean(self.selfishsystemstates)
            self.meanoptimalsystemstate = mean(self.optimalsystemstates)
            self.meansystemstate = mean([sum(k) for k in zip(self.selfishsystemstates, self.optimalsystemstates)])

            self.selfishwaitingtimes = []
            self.optimalwaitingtimes = []
            self.selfishservicetimes = []
            self.optimalservicetimes = []
            for p in self.completed:
                if p.arrivaldate >= warmup:
                    if type(p) is SelfishPlayer:
                        self.selfishwaitingtimes.append(p.waitingtime)
                        self.selfishservicetimes.append(p.servicetime)
                    else:
                        self.optimalwaitingtimes.append(p.waitingtime)
                        self.optimalservicetimes.append(p.servicetime)
            self.meanselfishwaitingtime = mean(self.selfishwaitingtimes)
            self.meanselfishsystemtime = mean(self.selfishservicetimes) + self.meanselfishwaitingtime
            self.meanoptimalwaitingtime = mean(self.optimalwaitingtimes)
            self.meanoptimalsystemtime = mean(self.optimalservicetimes) + self.meanoptimalwaitingtime

            self.selfishprobbalk = 0
            self.optimalprobbalk = 0
            for p in self.balked:
                if p.arrivaldate >= warmup:
                    if type(p) is SelfishPlayer:
```

```
                    self.selfishprobbalk += 1
                else:
                    self.optimalprobbalk += 1

        self.meanselfishcost = self.selfishprobbalk * self.costofbalking[1] + sum(self.selfishservicetimes) + sum(self.selfishwaitingtimes)
        self.meanoptimalcost = self.optimalprobbalk * self.costofbalking[1] + sum(self.optimalservicetimes) + sum(self.optimalwaitingtimes)
        self.meancost = self.meanselfishcost + self.meanoptimalcost
        if len(self.selfishwaitingtimes) + self.selfishprobbalk != 0:
            self.meanselfishcost /= self.selfishprobbalk + len(self.selfishwaitingtimes)
        else:
            self.meanselfishcost = False
        if len(self.optimalwaitingtimes) + self.optimalprobbalk != 0:
            self.meanoptimalcost /= self.optimalprobbalk + len(self.optimalwaitingtimes)
        else:
            self.meanselfishcost = False

        if self.selfishprobbalk + self.optimalprobbalk + len(self.selfishwaitingtimes) + len(self.optimalwaitingtimes) != 0:
            self.meancost /= self.selfishprobbalk + self.optimalprobbalk + len(self.selfishwaitingtimes) + len(self.optimalwaitingtimes)
        else:
            self.meancost = False

        if self.selfishprobbalk + len(self.selfishwaitingtimes) != 0:
            self.selfishprobbalk /= self.selfishprobbalk + len(self.selfishwaitingtimes)
        else:
            self.selfishprobbalk = False
        if self.optimalprobbalk + len(self.optimalwaitingtimes) != 0:
            self.optimalprobbalk /= self.optimalprobbalk + len(self.optimalwaitingtimes)
        else:
            self.optimalprobbalk = False

        sys.stdout.write("\n%sSummary statistics%s\n" % (10*"=",10*"="))

        sys.stdout.write("\n%sSelfish players%s\n" % (13*"-",10*"-"))
        sys.stdout.write("Mean number in queue: %.02f\n" % self.meanselfishqueuelength)
        sys.stdout.write("Mean number in system: %.02f\n" % self.meanselfishsystemstate)
        sys.stdout.write("Mean waiting time: %.02f\n" % self.meanselfishwaitingtime)
        sys.stdout.write("Mean system time: %.02f\n" % self.meanselfishsystemtime)
        sys.stdout.write("Probability of balking: %.02f\n" % self.selfishprobbalk)

        sys.stdout.write("\n%sOptimal players%s\n" % (13*"-",10*"-"))
        sys.stdout.write("Mean number in queue: %.02f\n" % self.meanoptimalqueuelength)
```

```
            sys.stdout.write("Mean number in system: %.02f\n" % self.
    meanoptimalsystemstate)
            sys.stdout.write("Mean waiting time: %.02f\n" % self.
    meanoptimalwaitingtime)
            sys.stdout.write("Mean system time: %.02f\n" % self.
    meanoptimalsystemtime)
            sys.stdout.write("Probability of balking: %.02f\n" % self.
    optimalprobbalk)

            sys.stdout.write("\n%sOverall mean cost (in time)%s\n" %
    (9*"-","-"))
            sys.stdout.write("All players: %.02f\n" % self.meancost)
            sys.stdout.write("Selfish players: %.02f\n" % self.meanselfishcost
    )
            sys.stdout.write("Optimal players: %.02f\n" % self.meanoptimalcost
    )
            sys.stdout.write(39 * "=" + "\n")


if __name__ == '__main__':
    import argparse
    parser = argparse.ArgumentParser(description="A simulation of an MM1 queue
     with a graphical representation made using the python Turtle module. Also,
     allows for some agent based aspects which at present illustrate results
    from Naor's paper: 'The Regulation of Queue Size by Levying Tolls'")
    parser.add_argument('-l', action="store", dest="lmbda", type=float, help='
    The arrival rate', default=2)
    parser.add_argument('-m', action="store", dest="mu", type=float, help='The
     service rate', default = 1)
    parser.add_argument('-T', action="store", dest="T", type=float, help='The
    overall simulation time', default=500)
    parser.add_argument('-p', action="store", dest="probofselfish", help='
    Proportion of selfish players (default: 0)', default=0, type=float)
    parser.add_argument('-c', action="store", dest="costofbalking", help='Cost
     of balking (default: False)', default=False, type=float)
    parser.add_argument('-w', action="store", dest="warmuptime", help='Warm up
     time', default=0, type=float)
    parser.add_argument('-s', action="store", dest="savefig", help='Boolean to
     save the figure or not', default=False, type=bool)
    inputs = parser.parse_args()
    lmbda = inputs.lmbda
    mu = inputs.mu
    T = inputs.T
    warmup = inputs.warmuptime
    savefig = inputs.savefig
    costofbalking = inputs.costofbalking
    if costofbalking:
        costofbalking = [inputs.probofselfish, inputs.costofbalking]
    q = Sim(T, lmbda, mu, speed=10, costofbalking=costofbalking)
    q.run()
    q.printsummary(warmup=warmup)
    q.plot(savefig)
```

References

[1] IEEE Research paper
https://ieeexplore.ieee.org/document/6552200

[2] m/m/1 queue code
https://introcs.cs.princeton.edu/python/43stack/mm1queue.py

[3] M/M/1 Queue simulation
https://github.com/sarthak0120/M-M-1-Queue-Simulation

[4] Simulation
https://github.com/drvinceknight/Simulating$_{Q}ueues/blob/master/MM1Q.py$