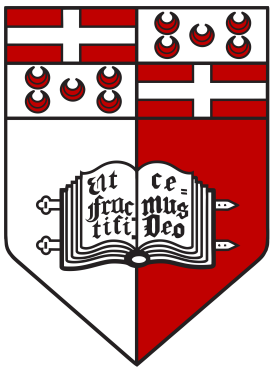


CCE2501: Modelling and Computer Simulation
-
Assignment

Full name: Miguel Dingli
I.D: 49997M
Course: B.Sc. (Hons) in Computing Science

Contents

1 Question 1 - Log-Normal Distribution

1.1 Description

To start with, any required modules were imported and some constants were defined. **tLimits** is an array containing the lower limit *0* and upper limit *5* of the interval $0 \leq t \leq 5$, while **mu** and **sigma** are the mean and standard deviation, respectively. Next, the log-normal distribution function was implemented in the form of a **lognorm** function which takes an input **t** and returns the equivalent of the function below:

$$y(t) = \frac{1}{t\sigma\sqrt{2\pi}} e^{\frac{(\ln t - \mu)^2}{2\sigma^2}} = \frac{\left(e^{\frac{(\ln t - \mu)^2}{2\sigma^2}}\right)}{t\sigma\sqrt{2\pi}}$$

```
import random
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

tLimits = [0, 5]
mu = 0
sigma = 0.75

def lognorm(t):
    return (np.exp(-((np.log(t) - mu) ** 2) / (2 * sigma ** 2))) / (t * sigma * np.sqrt(2 * np.pi))
```

Next, since the Von-Neumann accept-reject method requires a value M which is greater than $f(t)$ over $[0, 5]$, the Log-Normal distribution was generated using **lognorm** so that a suitable M value can be deduced from the values. The values were plotted.

```
tValues = np.arange(0.01, 5, 0.01)
yValues = np.array(list(map(lognorm, tValues)))
plt.figure()
plt.plot(tValues, yValues, '-b')
plt.ylabel('y')
plt.xlabel('t')
plt.title("Log-Normal Distribution")
plt.grid()
plt.show()
```

An **M** was now defined to be the *ceiling* rounding of the maximum *y-value* from the previously-generated values. Next, the **binRange** was defined using the difference between the two interval limits and the amount of bins, **binCount**, was set to 100. The array representing the bins **rand_lognorm** was defined to be an array of **binCount** zeros. Also, an array representing the bin limits, **bins**, was defined.

```
M = np.ceil(max(yValues))
print("M = %f" % M)

binRange = tLimits[1] - tLimits[0]
binCount = 100

rand_lognorm = np.zeros(binCount)           # Array of 'binCount' bins
bins = np.linspace(0, binRange, binCount)   # Array of 'binCount' bin limits from 0 to 'binRange'
```

The algorithm based on Von-Neumann's accept-reject method was now implemented as a python function **generatePoint** which generates a single point and adds it to the respective bin. A random number **u** is first generated from the uniform distribution based on $[0, 5]$ and then **v** is generated from the uniform distribution based on $[0, M]$. Next, if **v** is smaller than the distribution's *y-value lognorm(u)* for a *t-value* **u**, it is accepted. The first bin that the point's *t-value* **u** satisfies gets incremented by 1.

```
# Generate uniform random nos. and convert them to a lognormal dist. using accept-reject
def generatePoint():
    u = random.uniform(tLimits[0], tLimits[1])
    v = random.uniform(0, M)
    if v < lognorm(u):
        for j in range(0, len(bins)):
            if u < bins[j]:
                rand_lognorm[j] += 1
                break
```

The function is now called for 10,000 times to generate 10,000 points, and the array containing the generated points is then output.

```
for i in range(10000):
    generatePoint()

print(rand_lognorm)
```

Next, the probability density function will be plotted. Since the area under the curve formed by the counts in the bins is not equal to 1, the bin values will have to be scaled down by dividing them with the area under the curve which was calculated by finding the sum of the products of the fixed bin width and the bin values. The bin width is found by dividing the bin range by the number of bins. The probability density function is now plotted, using **bins** as the *x-axis* and the bin values divided by the area as the *y-axis*. The log-normal distribution produced using the standard Python library was also plotted.

```
binWidth = binRange / binCount
area = sum(binWidth * rand_lognorm)

plt.figure()
plt.plot(bins, rand_lognorm / area, 'bo', label="By Von-Neumann Method")
plt.plot(bins, stats.lognorm([sigma],loc=mu).pdf(bins), 'r-', label="By Standard Library")
plt.ylabel('Number of counts')
plt.xlabel('X, the log-normally distributed random variable')
plt.title("Probability Density Function")
plt.grid()
plt.legend()
plt.show()
```

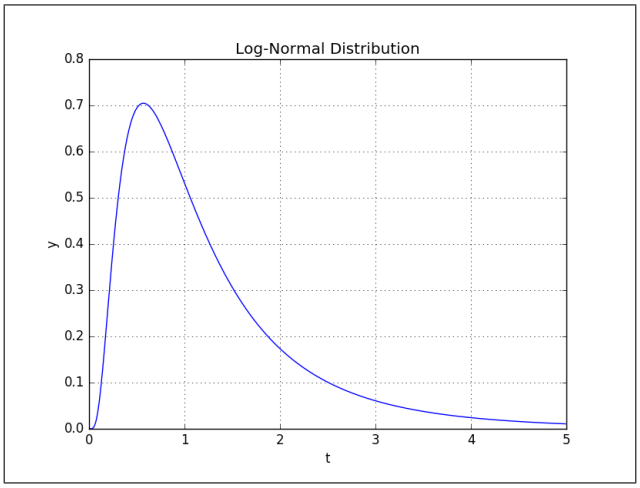
Kindly refer to the next page for the output.

1.2 Output

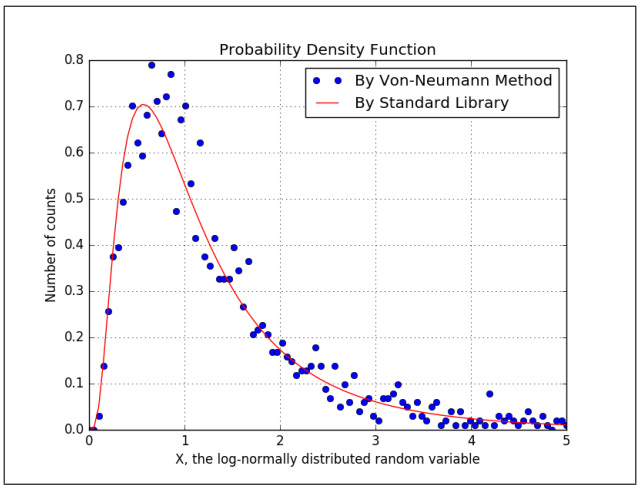
The following output shows that the rounded **M** value turned out to be 1. The output also shows the bin values for the points generated by the Von-Neumann method.

```
M = 1.000000
[  0.   0.   4.  13.  16.  32.  40.  43.  64.  76.  66.  74.  67.  61.  60.
  60.  82.  65.  53.  62.  56.  70.  45.  52.  31.  32.  37.  32.  41.  39.
  26.  21.  33.  23.  30.  28.  21.  21.  17.  21.  11.  18.  14.  16.  16.
  20.   8.   9.  10.   8.   8.  12.   8.   6.  11.   3.  13.   5.   5.   3.
   6.   5.   9.   2.   3.   2.   4.   6.   2.   7.   7.   5.   2.   3.   2.
   3.   5.   1.   3.   3.   1.   0.   3.   2.   1.   0.   0.   1.   3.   0.
   0.   2.   1.   1.   0.   3.   0.   4.   2.   2.]
```

Analysing the first plot, the **M** value of 1 is clearly greater than the log-normal distribution for all t values in the range $0 \leq t \leq 5$ The distribution seems to have a maximum *y-value* of around 0.7.



The second plot shows how, after the bin values were scaled down, the red curve and blue points trace, more or less, the same path. Obviously, if a greater number of iterations were to be performed, the accuracy of the blue points would greatly increase.



2 Question 2 - T-Junction Simulation

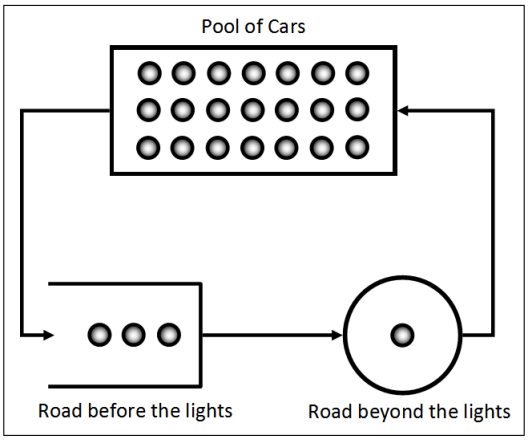
To perform a T-Junction simulation, two simulations based on the same model will be performed. Additionally, since two time ranges will be considered (morning and evening), a total of four simulations will be performed based on the same model.

2.1 Task A - Conceptual Model

The conceptual model described below is made up of a **pool of cars** that may eventually arrive at the **road before the lights** and then get *served* by the **road beyond the lights** when the lights are green.

- Pool of a virtually-unlimited amount of cars
  - Cars leave the pool all the time but leave more frequently at certain times and only if the road beyond the lights needs to be reached.
  - Cars are continuously operational and do not break down.
- Road before the lights (Queue)
  - **State variable: Lights** (Indicates whether lights are green or red).
  - The lights switch between green and red every 30-seconds
  - The inter-arrival time of cars depends on the time and the type of road.
  - A car is released only if the road's **Lights** are green, and server is not **Busy**.
  - Cars are sent for *service* in the order of arrival.
- Road beyond the lights (Server)
  - **State variable: Busy** (Indicates whether a car is passing or not).
  - Accepts one car at a time from the queue when **Lights** are green.
  - When a car starts service, it is released after exactly two seconds of service and so the server busy for a fixed two-second interval for each car.

The following is a simple system diagram of the model to be implemented:



2.2 Task B - GSL Code

In this section the GSL code implementing the conceptual model will be discussed. The value **theRoad** is a parameter to the model and is assumed to be available at runtime.

Initialize:

In the initialization, the fixed service time, lights change time, and simulation time limit are set to 2 seconds, 30 seconds, and 14,400 seconds (4 hours) respectively. Value **ROAD** is set to parameter *theRoad*. The current time **NOW**, queue length **Q** and the server busy state **S** are also initialised. The lights state **L** is set to green if the road to be simulated is *A*; otherwise it is set to red. This is done so that if the road is B, the lights are red in the first 30 seconds. Finally, a lights change event is scheduled at **LIGHTSCHANGETIME** from current time and an arrival event is scheduled at current time.

```
Initialize
    SERVICETIME = 2;        # fixed service time of 2 seconds
    LIGHTSCHANGETIME = 30;  # fixed lights change interval of 30 seconds
    TIMELIMIT = 14400       # fixed time limit of 4 hours (14400 seconds)
    ROAD = theRoad          # road that will be simulated
    NOW = 0;                # current time
    Q = 0;                  # queue length
    S = idle;               # server busy state
    IF ROAD == A THEN L = green ELSE L = red;   # starting lights state
    Schedule lights change event at time NOW;   # first lights change
    Schedule arrival event at time NOW;         # first car arrival
```

Main Loop:

The main loop will iterate until the maximum time limit is reached by **NOW**. When this is reached, the loop continues to iterate until the event list has only one event remaining; a lights change event. This allows all remaining events (excluding arrivals) to occur. In each iteration, the current time is updated to the first event's time and a routine is called based on the type of event. *Arrival* events are ignored if the time limit was exceeded.

```
WHILE NOW <= TIMELIMIT or event list has more than one item and DO
    NOW = time on the first event notice;
    IF event type == 'Arrival' and NOW <= TIMELIMIT THEN execute event routine arrival;
    IF event type == 'Begin Service' THEN execute event routine begin service;
    IF event type == 'End Service' THEN execute event routine end service;
    IF event type == 'Lights Change' THEN execute event routine lights change;
    Discard first event notice;
END WHILE
```

Event Routine Arrival:

In the arrival routine, the inter-arrival time is calculated (based on the time period [morning/evening] and road [A/B]) and an arrival is scheduled at **NOW** plus the inter-arrival time, for the next car to arrive. Since a car arrived, queue length **Q** is incremented. If the server is idle, then, if the road's lights are green a service start is scheduled at time **NOW** for service to start now. Alternatively, if the car is the only one in the queue, a service start is scheduled at the next time that the lights will change, so that the car gets immediately serviced when the lights change to green.

Kindly refer to the next page for the code.

```
Event Routine Arrival
    Schedule arrival event at time NOW + interarrival time;
    Q = Q + 1;
    IF S == idle THEN
        IF L == green THEN schedule begin service event at time NOW;
        ELSE IF Q == 1 THEN schedule begin service event at next lights change;
```

Event Routine Begin Service:

In the begin service routine, since a car left the queue, the queue length **Q** is decremented, and since the server is now servicing, its state **S** is set to busy. The car's end of service is then scheduled (using the fixed 2-second service time).

```
Event Routine Begin Service
    Q = Q - 1;
    S = busy;
    Schedule end service event at time NOW + SERVICETIME;
```

Event Routine End Service:

In the end service routine, since the server is no longer servicing, its state **S** is set to idle. Furthermore, if the queue has cars in it and the lights are green, a start of service is scheduled for the next car at current time.

```
Event Routine End Service
    S = idle;
    IF Q > 0 and L == green THEN schedule begin service event at time NOW;
```

Event Routine Lights Change:

In the lights change routine, the lights are inverted. Also, a lights change is scheduled at 30 seconds from current time, using the fixed **LIGHTSCHANGETIME**.

```
Event Routine Lights Change
    IF L == green THEN L = red ELSE L = green
    Schedule lights change event at time NOW + LIGHTSCHANGETIME;
```

2.3 Task C - Model Implementation

2.3.1 Description

To start with, the required modules were imported. Two convenience functions were now defined. **nowStr()** converts **NOW** to a form with hours and minutes, while **timeStr()** returns *Morning* if **morning** is *True*, and otherwise returns *Evening*.

```
from scipy import stats
from enum import IntEnum
import matplotlib.pyplot as plt
import time
import numpy as np

def nowStr():
    return time.strftime('%H h, %M min', time.gmtime(NOW))

def timeStr():
    return 'Morning' if morning else 'Evening'
```

Next, the three constants for service time, lights change time, and the time limit were defined. Two integer enums to represent the road type and event type were also defined.

```
# CONSTANTS
SERVICETIME = 2         # fixed service time of 2 seconds
LIGHTSCHANGETIME = 30   # fixed lights interval of 30 seconds
TIMELIMIT = 14400       # maximum time of 4 hours (14400 seconds)

class Road(IntEnum):
    A = 0               # Road A
    B = 1               # Road B

class EVT(IntEnum):
    ARRIVAL = 0         # Arrival event
    START_S = 1         # Start service event
    END_S = 2           # End service event
    LIGHTSCHANGE = 3    # Light change event
```

The following two functions are called whenever values that will be used for the required plots needs to be appended to the respective global arrays. **addPlot1Values()** corresponds to the queue length versus time plot, while **addPlot2Values()** corresponds to an arrivals versus time histogram that will be plotted for each road and time period.

```
def addPlot1Values():
    global plot1
    plot1[0].append(NOW / 3600)     # current time
    plot1[1].append(qLen)           # car count in service node

def addPlot2Values():
    global plot2
    plot2.append(NOW / 3600)        # time of arrival
```

The next function first generates a time value in the range $1 \leq t \leq 10$ from an exponential distribution by looping until a value satisfying the range is found. The value is then divided by the normal probability density function's y-value at the time in hours $\left(\frac{\textbf{NOW}}{3600}\right)$ and using a global mean and standard deviation based on current **road**. The second function **scheduleEvent** appends an event to the event list based on the arguments.

```
def getInterarrivalTime():
    iat = 0
    while iat < 1 or iat > 10:
        iat = (np.random.exponential(2))  # generate iat from exponential distribution
    return iat / stats.norm.pdf(NOW / 3600, loc=iatMeans[road], scale=iatStdevs[road])

def scheduleEvent(eventType, eventTime):
    eventList.append([eventType, eventTime])
```

The first event routine to be discussed is the arrival routine. Firstly, an inter-arrival time is obtained, and an arrival for the next car is scheduled. The global values **totalARR** (total arrivals) and **totalIAT** (total inter-arrival times) are incremented accordingly. Next, similar as to was discussed in the GSL, the queue lenth is incremented and a start of service event is scheduled if the server is not busy, and if either the lights are green, or this car is the only car in the queue. If the latter is the case, **nextLightChange** is set to the time of the first event in the event list which is a lights change event. Finally, the plot values are added and event details are written to a global file for logging purposes.

```
52 def arrival():
53     global totalARR, totalIAT, qLen
54 
55     # for the next car...
56     iat = getInterarrivalTime()
57     scheduleEvent(EVT.ARRIVAL, NOW + iat)  # schedule arrival of next car
58     totalARR += 1       # 1 more car arrived
59     totalIAT += iat     # added to total inter-arrival times
60 
61     # for the current car...
62     qLen += 1
63     if not serverBusy: # schedule service start of current car if server is not busy
64         if greenLights:
65             scheduleEvent(EVT.START_S, NOW)
66         elif qLen == 1:  # if only 1 car, schedule its service to closest lights change
67             nextLightChange = next(event[1] for event in eventList if event[0] == EVT.LIGHTSCHANGE)
68             scheduleEvent(EVT.START_S, nextLightChange)
69 
70     # add plot values and output to file
71     addPlot1Values()  # for queue length vs. time plot
72     addPlot2Values()  # for time of arrival
73     FILE.write('Car arrived.   (QLen: %d) (Time: %s) [Next in %f time]\n' % (qLen, nowStr(), iat))
```

In the start of service routine, queue length is decremented, server state is set to busy, and an end of service is scheduled at two seconds from **NOW**. Plot values are then added and the event details are written to the global file.

```
75 def startservice():
76     global qLen, serverBusy
77 
78     qLen -= 1                                  # car left the queue
79     serverBusy = True                          # server now busy
80     scheduleEvent(EVT.END_S, NOW + SERVICETIME)  # schedule service end of current car
81 
82     # add plot values and output to file
83     addPlot1Values()  # for queue length vs. time plot
84     FILE.write('Service start. (QLen: %d) (Time: %s)\n' % (qLen, nowStr()))
```

In the end of service routine, server state is set to idle (i.e. busy is *False*), and if there is a car in the queue and the lights are green, a start of service is scheduled for the next car. Plot values are then added and the event details are written to the global file.

```
86 def endservice():
87     global serverBusy
88 
89     serverBusy = False                     # server now free
90     if qLen > 0 and greenLights:
91         scheduleEvent(EVT.START_S, NOW)     # schedule next car service
92 
93     # add plot values and output to file
94     addPlot1Values()  # for queue length vs. time plot
95     FILE.write('Service end.   (QLen: %d) (Time: %s)\n' % (qLen, nowStr()))
```

The final event routine is for the changing of the lights. The boolean **greenLights** is simply set to its inverse and a change of lights event is scheduled at 30 seconds from now.

```
97  def lightschange():
98      global greenLights
99      greenLights = not greenLights
100     scheduleEvent(EVT.LIGHTSCHANGE, NOW + LIGHTSCHANGETIME)  # schedule lights change
101
102     # output to file
103     FILE.write('Lights change. (QLen: %d) (Time: %s)\n' % (qLen, nowStr()))
```

The function that will initialize most values takes two parameters; the road type and a file-name suffix for the output file (excluding the file type *.txt*). **NOW** is initialized to zero, the **eventList** is set to empty, and global **road** is set to the function's **theRoad** argument. Queue length is initialized to zero, the server's state is set to idle, and the lights are set to green (i.e. *True*) if the road to be simulated is road A since the lights for this road are initially green while those for road B are initially red. The plot arrays are set to empty, and the two total variables are set to zero. A file that will be used globally to write event details to is opened. Finally, a car arrival is scheduled at current time, and a change of lights is scheduled at 30 seconds from now.

```
107 def initialize(theRoad: Road, fileSuffix: str):
108     global NOW, eventList, qLen, serverBusy, greenLights, road
109     global plot1, plot2, totalARR, totalIAT, FILE
110
111     NOW = 0                          # current time
112     eventList = []                   # [eventType, eventTime]
113     road = theRoad                   # the road
114
115     qLen = 0                         # length of car queues
116     serverBusy = False               # indicates if server is busy; initially idle
117     greenLights = (road == Road.A)   # indicates if road is green (T) or red (F)
118
119     plot1 = [[], []]                 # values for car count vs. time plot
120     plot2 = []                       # values of time of arrivals
121
122     totalARR = 0                     # total arrivals
123     totalIAT = 0                     # total inter-arrival time
124
125     # open file to log event details to
126     FILE = open('task2_Output_' + fileSuffix + '.txt', 'w')
127
128     scheduleEvent(EVT.ARRIVAL, NOW)  # arrival of 1st car
129     scheduleEvent(EVT.LIGHTSCHANGE, NOW + LIGHTSCHANGETIME)  # event for 1st lights change
```

In the function that will run the simulation, the main simulation loop can be found. As explained in the GSL, the loop will loop until the time limit is reached, and the event list only has one item. For each iteration, the event list is sorted and the first event is popped. **NOW** is updated to the event's time, and the appropriate routine is called based on the type of the event. If the event is an arrival and the time limit was exceeded by the event, then the arrival does not occur. Other events are allowed to occur beyond the time limit, until the event list gets down to one event (a change of lights event).

After the loop is done, the global file is closed and any final calculations are performed, the results of which are output. This includes the service rate, which is the reciprocal of the constant two-second service time, and the arrival rate, which was set as following:

$$\text{Arrival rate} = \frac{1}{\bar{r}}; \text{...where...} \bar{r} = \frac{1}{n}\sum_{i=1}^{n} r_i = \frac{\textbf{totalIAT}}{\textbf{totalARR}}$$

The outputs also include a calculation for the traffic intensity, as required, which is calculated as follows:

$$\text{Traffic intensity} = \frac{1/\bar{r}}{1/\bar{s}} = \frac{\textbf{arrivalRate}}{\textbf{serviceRate}}$$

At the end of the function, the two arrays of plot values **plot1** and **plot2** are returned.

```
def runSimulation() -> ([[int], [int]], [float]):
    global NOW, eventList

    # Note: event list contains at least 1 event, a change lights event
    while NOW <= TIMELIMIT or len(eventList) > 1:
        eventList = sorted(eventList, key=lambda x: x[1])      # sort by event time
        firstEvent = eventList.pop(0)                          # get first event
        NOW = firstEvent[1]                                    # update time
        if firstEvent[0] == EVT.ARRIVAL and NOW <= TIMELIMIT:  # no more arrivals after limit
            arrival()  # car arrived
        elif firstEvent[0] == EVT.START_S:
            startservice()  # start car service
        elif firstEvent[0] == EVT.END_S:
            endservice()  # end car service
        elif firstEvent[0] == EVT.LIGHTSCHANGE:
            lightschange()  # lights swap

    # close event details file
    FILE.close()

    # calculations
    arrivalRate = totalARR / totalIAT   # arrival rate
    serviceRate = 1 / SERVICETIME       # service rate
    print('Calculations for Road ' + ('A ' if road == Road.A else 'B ') + '(' + timeStr() + '):')
    print('\tArrival rate: %f' % arrivalRate)
    print('\tService rate: %f' % serviceRate)
    print('\tTraffic intensity: %f' % (arrivalRate / serviceRate))
    print()

    return plot1, plot2
```

After the two arrays **plot1** and **plot2** are computed for both of the two roads, they are eventually fed into the function **plots()** such that, for example, *plt1[Road.A]* gives the **plot1** values for road A. This function plots the queue length versus time for both roads in a time period and also plots a car arrival counts versus time histogram.

```
def plots(plt1, plt2):
    # qLen vs. time plot
    plt.figure()
    plt.plot(plt1[0][0], plt1[0][1], '-bo', label='Road A')
    plt.plot(plt1[1][0], plt1[1][1], '-ro', label='Road B')
    plt.ylabel('Number of cars in queue')
    plt.xlabel('Time')
    plt.title('Queue length as a function of time (' + timeStr() + ')')
    plt.legend()
    plt.grid()

    # arrivals vs. time plot
    plt.figure()
    plt.hist(plt2[0], bins=20, range=[0,4], histtype='step', color='blue', label='Road A')
    plt.hist(plt2[1], bins=20, range=[0,4], histtype='step', color='red', label='Road B')
    plt.ylabel('Car arrivals')
    plt.xlabel('Time')
    plt.title('Arrivals (' + timeStr() + ')')
    plt.legend()
    plt.grid()
```

The final function to be implemented is **fullSimulationWithPlots()**, which performs two simulations (one for each road) for a particular time period (Morning/Evening). This function takes a boolean time period (Morning if *True*), and the means and standard deviations to be used in the inter-arrival time generation in the two simulations. The function initializes the global value **morning** and the global arrays **iatMeans** and **iatStdevs**. Then, for two times, the initialize function is first called for road A and the simulation is then run. The plot values returned from the simulation are passed to **plots()**.

```
185 def fullSimulationWithPlots(timePeriod: bool, means: [float, float], stdevs: [float, float]):
186     global morning, iatMeans, iatStdevs
187     morning = timePeriod
188     iatMeans = means
189     iatStdevs = stdevs
190 
191     initialize(Road.A, timeStr() + '_A')             # Initialize
192     (plot1_1, plot2_1) = runSimulation()             # Road A
193     initialize(Road.B, timeStr() + '_B')             # Initialize
194     (plot1_2, plot2_2) = runSimulation()             # Road B
195     plots((plot1_1, plot1_2), (plot2_1, plot2_2))    # Plot
```

To run the simulation for four times, the last function discussed is called twice; once for Morning (*True*) and once for Evening (*False*). The means for the inter-arrival time generation (2, 2.5, 2, and 1.5) were derived by subtracting the time period's start time (e.g. 6:30am for Morning) from the given mean (e.g. 8:30am for road A), which sets the mean of the normal distribution at the hour where arrivals will be most frequent (e.g. at hour 2), starting from time zero. The standard deviations (1, 0.95, 0.95, 1) can remain the same as given since they are relative to the mean.

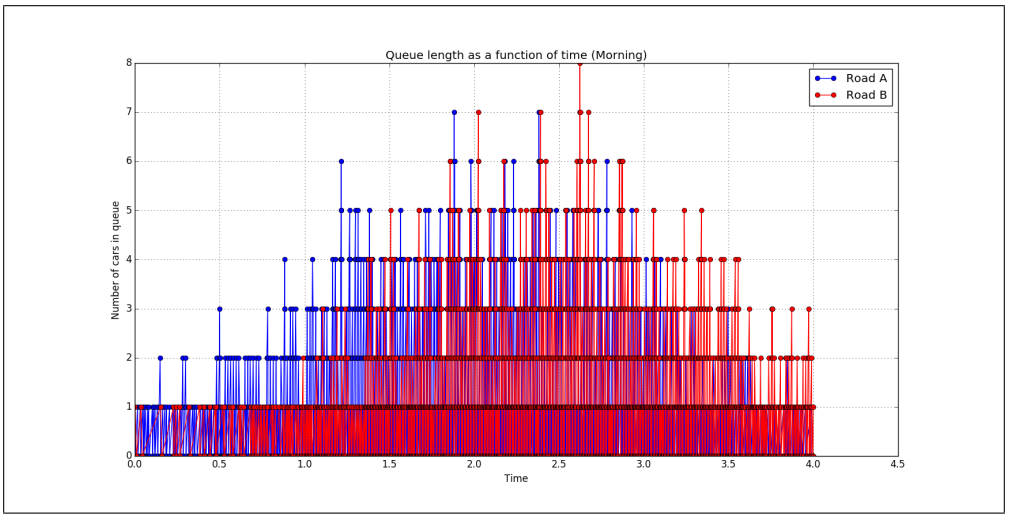
```
197 fullSimulationWithPlots(True, [2, 2.5], [1, 0.95])   # Morning
198 fullSimulationWithPlots(False, [2, 1.5], [0.95, 1])  # Evening
199 plt.show()  # show plots
```

2.3.2 Output

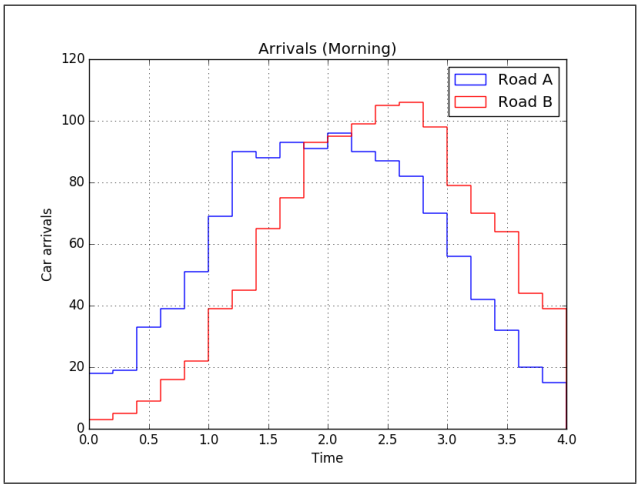
The following output shows the answers to the calculations performed for each road, for each time period. The arrival rates are all identical, up to two decimal places, and since the service rate is fixed to 0.5, the traffic intensities are also identical, up to two d.p.

```
Calculations for Road A (Morning):
        Arrival rate: 0.081723
        Service rate: 0.500000
        Traffic intensity: 0.163447

Calculations for Road B (Morning):
        Arrival rate: 0.081192
        Service rate: 0.500000
        Traffic intensity: 0.162384

Calculations for Road A (Evening):
        Arrival rate: 0.081863
        Service rate: 0.500000
        Traffic intensity: 0.163726

Calculations for Road B (Evening):
        Arrival rate: 0.078926
        Service rate: 0.500000
        Traffic intensity: 0.157852
```

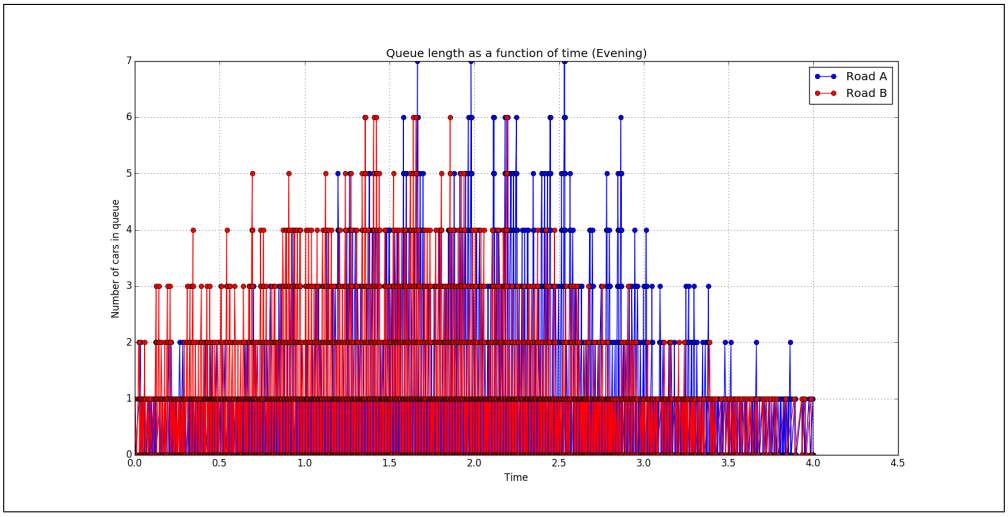
The first plot output is the queue length versus time for the Morning time period. The plot shows how for both roads, the queue length peaks increase during the first half and decrease in the second due to a normal distribution used for the inter-arrival time generation. Due to a lower mean for road A's inter-arrival time generation, road A has higher queue length peaks in the first 2 hours while road B has higher queue length peaks in the last $1\frac{1}{2}$ hours. Road B has a narrower span of higher peaks than road A due to road B's mean (9:00 am) being greater than the mid-way (8:30 am) of the 4-hour simulation (6:30 am to 10:30 am). Also, for road B, the transition between increasing and decreasing queue length peaks happens later, due to a larger mean.



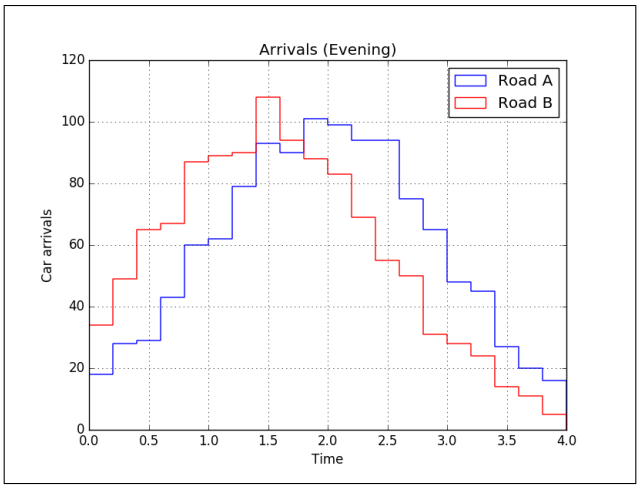
The second plot output is a histogram of the car arrivals versus time for the Morning time period. From this histogram, it can be deduced that for road A, the maximum amount of car arrivals is around the 2-hour period (c.95 arrivals between 2.0 and 2.2), which matches with the 8:30 am mean for inter-arrival time generation. For road B, the maximum amount of car arrivals is around the $2\frac{1}{2}$-hour period (c.105 arrivals between 2.6 and 2.8), which also matches with the 9:00 am mean, since the starting time was 6:30 am.



The third plot output is the queue length versus time for the Evening time period. Since road A now has a higher mean than road B for the inter-arrival time generation, road B has higher queue length peaks in the first $1\frac{1}{2}$ hours while road A has higher queue length peaks in the last 2 hours. Road B has a narrower span of higher peaks than road A due to road B's mean (5:30 pm) being less than the mid-way (6:00 pm) of the 4-hour simulation (4:00 pm to 8:00 pm). Also, for road B, the transition between increasing and decreasing queue length peaks happens earlier, due to a smaller mean.



The fourth plot output is a histogram of the car arrivals versus time for the Evening time period. From this histogram, it can be deduced that for road A, the maximum car arrivals is around the 2-hour period (c.102 arrivals between 1.8 and 2.0), which matches with the 6:00 pm mean for inter-arrival time generation. For road B, the maximum amount of car arrivals is around the $1\frac{1}{2}$-hour period (c.110 arrivals between 1.4 and 1.6), which also matches with the 5:30 pm mean, since the starting time was 4:00 pm.



2.4 Task D - Differences Between the Two Roads

After analysing the outputs, some deductions can be made with respect to the difference between roads A and B in the morning and evening:

- In the morning, road A peaked in the amount of car arrivals at around 8:30 am, while road B peaked not long after, at around 9:00 am.
- In the evening, road B peaked in the amount of car arrivals at around 5:30 pm, while road A peaked not long after, at around 6:00 pm.
- In general, apart from when the peaks occur, the roads show similar behaviour when it comes to overall arrival rate and traffic intensity. Even though road A had a higher traffic intensity and that road B had a higher maximum arrival count in both Morning and Evening, the differences were not seen as significant enough for conclusions to be formed.

To reduce the overall traffic, the transport authorities can implement a system where the interval between the switching of lights is dynamic, rather than fixed. At the times when road A is expected to receive *more* cars than road B, the lights should remain green longer for road A than for road B, so that more cars can pass when the lights are green. Similarly, when road A is expected to receive *less* cars than road B, the lights should remain green longer for road B than for road A.

The transport authorities can also simply promote the use of alternate means of transport, such as cycling, and work towards the required infrastructure. For each person that decides to use a bicycle rather than a car, there may be one less car arriving at the lights.

3 Question 3 - Monte Carlo Simulation

3.1 Task A - Area Under the Curve

3.1.1 Description

For this question, three module were imported. To start with, a function **graph(x)** was defined to take an x-value and return the y-value according to the given polynomial:

$$y(x) = 5x^4 - 8.7x^3 + 33x^2 + 21x + 10.8$$

For the Monte Carlo simulation to be performed, the maximum y value of the plot over the given x range is required. For this, a set of 10,000 x-values in the range $0 \leq x \leq 5$ and corresponding y-values were obtained and plotted. From the output of the plot (presented in the *Output* section), it was deduced that in the range of x-values considered, the maximum y-value is at the point where $x = 5$.

```
1  import random
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  def graph(x):
6      return 5 * (x ** 4) - 8.7 * (x ** 3) + 33 * (x ** 2) + 21 * (x) + 10.8
7
8  xValues = np.linspace(0, 5, 10000)
9  yValues = np.array(list(map(graph, xValues)))
10
11 plt.figure()
12 plt.plot(xValues, yValues, '-b')
13 plt.xlabel("x")
14 plt.ylabel("y")
15 plt.title("Graph of y = 5(x^4) - 8.7(x^3) + 33(x^2) + 21(x) + 10.8")
16 plt.grid()
17 plt.show()
```

Two arrays representing the x and y limits were now defined using previous values. Since the area under the graph will be found, the y lower limit was taken to be zero. Two values representing the range of the x axis and y axis were also computed. Additionally, two arrays that will hold the points under and over the graph, respectively, were defined.

```
21 xLimits = [0, 5]
22 yLimits = [0, graph(5)]
23 xRange = xLimits[1] - xLimits[0]
24 yRange = yLimits[1] - yLimits[0]
25
26 pointsUnderGraph = []
27 pointsOverGraph = []
```

The function that will perform the simulation was now implemented. The function takes a number of **iterations** to be performed, and a boolean **appendToArray** indicating whether the generated points should be appended to the pair of points arrays (**pointOverGraph** and **pointsUnderGraph**). The count of points under the graph is initialized to zero. In each iteration of the following *for* loop, a point with x and y-values within the previously-defined limits is generated using a uniform distribution. **isUnderGraph** is now set to *True* if the point is under the graph (i.e. its y value is less than the graph's y value obtained by calling **graph()** with the point's x value). The count is now accordingly incremented based on whether the point is below the graph or not and if **appendToArray** is *True*, the point is appended to the appropriate array.

After the loop is done, the area under the curve is computed by first finding the ratio of the points under the graph to the number of points generated and multiplying it by the area of the whole rectangular space defined by **xRange** and **yRange**. Hence, a fraction of this area (defined by the points) is taken as an estimate of the area under the graph.

```
29 def calcArea(iterations: int, appendToArray: bool) -> float:
30     countUnder = 0
31
32     for i in range(iterations):
33         point = [random.uniform(xLimits[0], xLimits[1]), random.uniform(yLimits[0], yLimits[1])]
34         isUnderGraph = point[1] < graph(point[0])
35         countUnder += 1 if isUnderGraph else 0
36
37         if appendToArray:
38             if isUnderGraph:
39                 pointsUnderGraph.append(point)
40             else:
41                 pointsOverGraph.append(point)
42
43     return (countUnder / iterations) * (xRange * yRange)
```

To perform 100,000 iterations, the **calcArea** function is called with parameters 100,000 and *True* (since the actual points will be used). The result gets stored in **area**. For the analytically-achieved area, integration was performed from 0 to 5 on the polynomial:

$$\begin{aligned}\int_0^5 \left(5x^4 - 8.7x^3 + 33x^2 + 21x + 10.8\right) &= \left[\frac{5x^5}{5} - \frac{8.7x^4}{4} + \frac{33x^3}{3} + \frac{21x^2}{2} + 10.8x\right]_0^5 \\ &= (5)^5 - 2.175(5)^4 + 11(5)^3 + 10.5(5)^2 + 10.8(5) \\ &= 3125 - 1359.375 + 1375 + 262.5 + 54 \\ &= 3457.125\end{aligned}$$

The two areas were then output, along with the percentage error.

```
47 area = calcArea(100000, True)
48 analytical = 3457.125
49 print("Area after 100,000 iterations: %f" % area)
50 print("Area achieved analytically:    %f" % analytical)
51 print("Percentage error:              %f" % np.abs(100 * (analytical - area) / analytical))
```

The resultant points generated when the **calcArea** was called with a *True* value are now found in **pointsUnderGraph** and **pointsOverGraph**. These arrays are converted to *numpy* arrays for convenience and are then plotted as a scatter plot which will distinguish between the points over the graph and the points under it by colour.

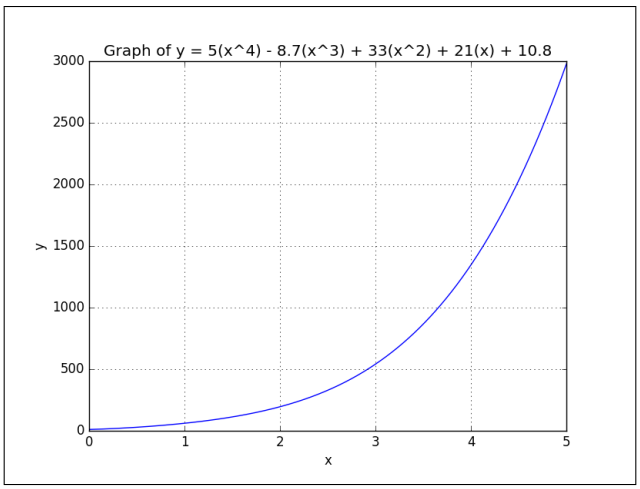
```
53 np_underGraph = np.array(pointsUnderGraph)
54 np_overGraph = np.array(pointsOverGraph)
55
56 plt.figure()
57 plt.scatter(np_underGraph[:, 0], np_underGraph[:, 1], c='b', label="Under Graph")
58 plt.scatter(np_overGraph[:, 0], np_overGraph[:, 1], c='r', label="Over Graph")
59 plt.title("Plot of Random Points Under and Over the Graph")
60 plt.legend()
61 plt.grid()
62 plt.show()
```

3.1.2 Output

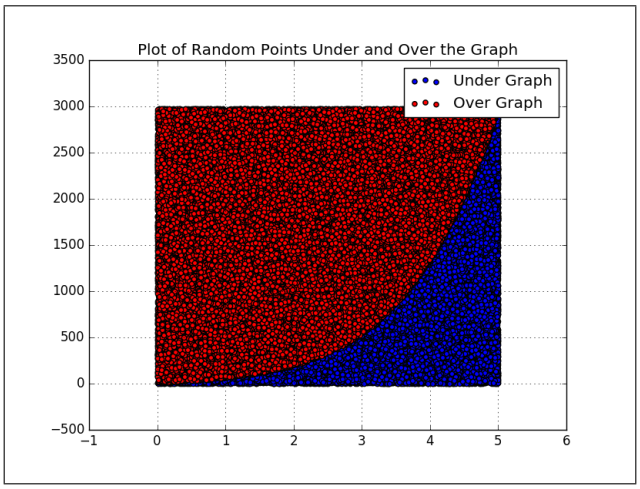
The following output shows how after 100,000 iterations, the area estimate of 3432.788580 came close to the area 3457.125 achieved analytically. However, it still leaves more accuracy to be desired, which will be achieved with more iterations.

```
Area after 100,000 iterations: 3432.788580
Area achieved analytically:    3457.125000
Percentage error:              0.703950
```

The following output is a graphical representation of the given polynomial:



The final output shows the 100,000 points that were generated by **calcArea** in the form of a scatter plot. The points that turned out to be under the graph were plotted in blue, while the ones which turned out over the graph were plotted in red.



3.2 Task B - More Iterations

3.2.1 Description

Continuing on the previous code, the simulation will now be run with an increasing number of iterations. The numbers of iterations were defined using **range**, starting from 10,000, up to 1,000,001 (+1 so that 1,000,000 is included in the resultant set of values), with a step of 10,000. An array **areaValues** which will hold the areas to be computed by the **calcArea** is defined as empty.

Now, the simulation is performed for each value *N_iter* in **N_iters** as the amount of iterations. The resultant areas area appended to the array of areas. After the simulation is run for various times up to 1,000,000 iterations, the final area is taken as the average of all the areas computed, using *numpy*'s **mean()** function. The average area, the previous analytically-achieved area, and the percentage error are now output.

```
66 N_iters = range(10000, 1000001, 10000)
67 areaValues = []
68 print()
69 for N_iter in N_iters:
70     areaValues.append(calcArea(N_iter, False))
71     print("Area after %d iterations: %f" % (N_iter, areaValues[-1]))
72 
73 averageArea = np.mean(areaValues)
74 print()
75 print("Average area:                %f" % averageArea)
76 print("Area achieved analytically:  %f" % analytical)
77 print("Percentage error:            %f" % np.abs(100 * (analytical - averageArea) / analytical))
```

A plot of the areas of the curve achieved through Monte Carlo simulation versus the number of iterations is now plotted. On the same plot, a constant line representing the area achieved analytically was included, using the minimum and maximum number of iterations as the x-values, and the area as the corresponding y-value for both x-values.

```
79 plt.figure()
80 plt.plot(N_iters, areaValues, '-bx')
81 plt.plot([N_iters[0], N_iters[-1]], [analytical, analytical], '-r')
82 plt.xlabel("Number of iterations")
83 plt.ylabel("Area")
84 plt.title("Plot of Area versus No. of Iterations")
85 plt.grid()
86 # plt.show()
```

The error values (an array of percentage errors) were obtained by applying the percentage error formula below to each area value:

$$PercentageError = 100 \times \frac{|\text{estimated area} - \text{actual area}|}{\text{actual area}}$$

The percentage errors were then plotted against the number of iterations, with a superimposed constant line representing the 1% mark. The minimum number of iterations required to achieve a value for the area beyond which the area remains within 1% of the analytical value will now be found and stored in **minimumIter**. The array of error values is traversed in reverse to find the first case for which the error is greater than 1%. Once this is found, the minimum iterations is set to the next corresponding number of iterations by index (in correct order), which is the required minimum number of iterations after which all errors are less than 1%. This value is then output.

Kindly refer to the next page for the code.

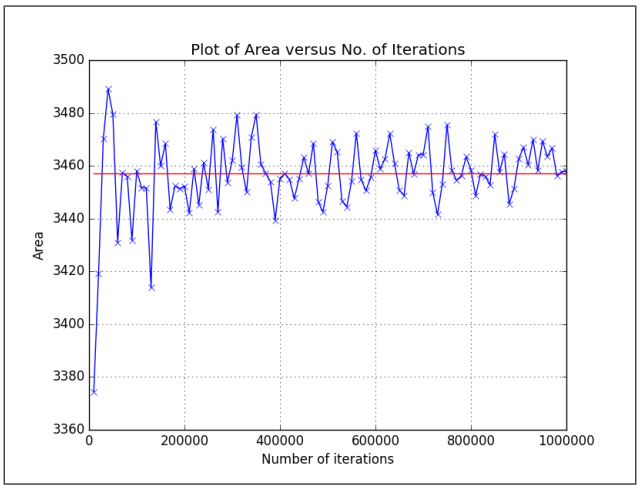
```
errorValues = list(map(lambda a: 100 * np.abs(a - analytical) / analytical, areaValues))

plt.figure()
plt.plot(N_iters, errorValues, '-bx')
plt.plot([N_iters[0], N_iters[-1]], [1, 1], '-r')
plt.xlabel("Number of iterations")
plt.ylabel("Percentage Error")
plt.title("Plot of Percentage Error versus No. of Iterations")
plt.grid()
plt.show()

minimumIter = 0
for i in reversed(range(len(errorValues))):  # traverse in reverse
    if errorValues[i] > 1:
        minimumIter = N_iters[i+1]  # at i+1 was last for which error was < 1%
        break
print()
print("Minimum number of iterations: %d" % minimumIter)
```

3.2.2 Output

The following output shows only the start and end parts of the outputs indicating the area computed after a number of iterations. The output also shows the average of the areas calculated, the area achieved analytically, and the percentage error, which is now much less than when only 100,000 iterations were performed. Finally, the output also shows that the minimum number of iterations required such that the percentage error remains below 1% is 140,000 in this particular case.

```
Area after 10000 iterations: 3374.413900
Area after 20000 iterations: 3419.088400
Area after 30000 iterations: 3470.215883
Area after 40000 iterations: 3489.078450
Area after 50000 iterations: 3479.547890
...
Area after 960000 iterations: 3463.468172
Area after 970000 iterations: 3466.756552
Area after 980000 iterations: 3456.241173
Area after 990000 iterations: 3457.761174
Area after 1000000 iterations: 3458.014781

Average area:                3456.633886
Area achieved analytically:  3457.125000
Percentage error:            0.014206

Minimum number of iterations: 140000
```

The first plot output shows the plot of the different values for the area of the curve achieved through *MC* simulation versus the number of iterations performed, along with a superimposed constant line representing the area achieved analytically. It can be observed that initially, the estimate is very inaccurate, but immediately converges to approximately 3460 ± 20 after around 150,000 iterations.



Kindly refer to the next page for the second plot.

The second plot output is that of the percentage error versus the number of iterations with a superimposed constant line highlighting the 1% error mark. From this plot, it is clear that the estimated minimum number of iterations required such that the error percentage remains below 1% of 140,000 (as seen in the first output) is correct. Beyond 140,000 iterations, the percentage error clearly stays below the 1% error mark.