Computer Modelling and Simulation Continuous Assessment

700074708

Contents

1 Introduction

We are simulating two queueing approaches, M/M/C/C and M1+M2/M/C/C, representing a wireless cell and and a wireless cell with handover respectively. However, these details are an abstraction and do not affect the implementation procedures that are about to be covered.

An M/M/C/C queueing system is a mathematical model used to analyse queuing or waiting line behaviour in various scenarios, such as call centres or service centres. It's characterised by two key features: Markovian Arrivals (M) where arrivals follow exponential distribution and are independent, and Markovian Service (M) where service times are exponentially distributed and independent. The "C/C" component indicates a finite number of service channels (C) and a finite customer population (C). This model helps predict system performance metrics, like average queue length and customer wait times, making it valuable for optimising resource allocation and capacity planning in real-world applications. In contrast, an M1+M2/M/C/C system allows for two distinct service rates (M1 and M2) and is still characterised by a finite number of servers and a finite customer population.

2 Validation formulae

It is important that we define the key analytical formula used to compare our models to.

2.1 MMCC Analytical Formulae

We examine some analytical approaches to determining key metrics. The following are key definitions:

- $\lambda =$ arrival rate
- $\mu =$ service rate
- $c =$ number of servers
- $A =$ traffic intensity, and is calculated by $\frac{\lambda}{\mu}$

Blocking Probability: The Erlang B formula is a fundamental concept in queueing theory, primarily used to calculate the blocking probability in a telephone exchange system or any system with a fixed number of resources (e.g. servers) and no waiting space. It determines the likelihood that a new call (or customer) will be rejected because all resources are already occupied. This formula is crucial in designing and optimising telecommunication systems, as it helps balance the need for sufficient capacity to handle call volumes against the cost of providing excess resources. By using the Erlang B formula, planners can estimate the number of lines required to achieve a desired level of service quality without excessive blocking or resource wastage. $B(c,A)$ represents the probability of blocking, given c servers and some traffic intensity A.

$$B(c,A) = \frac{\frac{A^c}{c!}}{\sum_{k=0}^{c} \frac{A^k}{k!}} \tag{1}$$

Server Utilisation: In queueing systems, particularly in scenarios like the M/M/C/C model, server utilisation refers to the average proportion of time that servers are actively serving customers. It's a key metric used to assess the efficiency of resource usage in systems where multiple servers or service channels are involved, like call centres or network servers. The server utilisation formula helps in determining how effectively the servers are being used and is critical for capacity planning and performance analysis. By understanding server utilisation,

managers can make informed decisions about scaling resources up or down to meet demand, optimise service levels, and control costs. High utilisation implies servers are frequently busy, while low utilisation indicates underused resources. The following describes the server utilisation proportion U.

$$U = \frac{\lambda \times (1 - B)}{c \times \mu} \tag{2}$$

Steady State Formula: The steady-state probabilities are essential in queueing theory because they provide the long-term probabilities of the system being in different states, assuming the system has been running for a sufficient length of time to reach equilibrium. In an M/M/C/C queue, these probabilities are crucial for understanding how often the system is fully occupied, how frequently customers are blocked, and for calculating key performance metrics like the average number of customers in the system or the blocking probability. They are fundamental in designing and analysing systems with limited capacity, such as call centres, server farms, or any system where arriving customers (or requests) might be turned away when the system is at full capacity. These probabilities are needed to calculate the average number of customers in the system. $P_n =$ is the steady state probability given for n servers.

$$P_0 = \left(\sum_{k=0}^{C} \frac{A^k}{k!} \right)^{-1} \tag{3}$$

$$P_n = \frac{\frac{A^n}{n!}}{\sum_{k=0}^{c} \frac{A^k}{k!}} \tag{4}$$

Average Number of Customers in System: The average number of customers in the system is a crucial metric in queueing theory. It's used to understand how busy the system is on average and to assess the system's performance. In practical terms, this metric helps in designing and optimising systems such as call centres, network routers, and customer service desks, where it's important to balance customer demand with available resources. Knowing the average number of customers can guide decisions on how many servers are needed to maintain an acceptable level of service without excessive waiting or resource under-utilisation. This metric is particularly relevant in systems with no queues, like the M/M/C/C model, where arriving customers are blocked or lost if all servers are busy.

$$L = \sum_{n=0}^{c} n \times P_n \tag{5}$$

2.2 MMCC actual formulae

On completion of a simulation, the following formula are used to calculate key metrics using real data. These can be used in comparison with the analytical results.

Blocking probability:

$$B(c,A)_{actual} = \frac{totalLoss}{totalArrivals}$$

Server utilisation:

$$U_{actual} = \frac{areaServerStatus}{c * simTime}$$

Average number of customers in system:

$$L_{actual} = \frac{areaServerStatus}{simTime}$$

2.3 M1M2MCC Analytical Formulae

We must redefine some key terms in our models:

- λ_1 = arrival rate for new call
- λ_2 = arrival rate for handover call
- A_1 = traffic intensity for both λ_1 and λ_2, calculated by $\frac{\lambda_1+\lambda_2}{\mu}$
- A_2 = traffic intensity for λ_2, calculated by $\frac{\lambda_2}{\mu}$

Steady state probability: This approach is extended and adapted from the MMCC approach, and is used to calculate the steady state probabilities for some state k:

$$P_0 = \left(\sum_{k=0}^{C-N} \frac{(\lambda_1+\lambda_2)^k}{\mu^k k!} + \sum_{k=C-N+1}^{C} \frac{(\lambda_1)^k}{\mu^k k!} \right)^{-1}$$

$$P(k) = \begin{cases} \frac{1}{k!}\left(\frac{\lambda_1+\lambda_2}{\mu}\right)^k P_0, & \text{for } 0 \le k \le C-N, \\ \frac{1}{k!}\left(\frac{\lambda_1+\lambda_2}{\mu}\right)^{C-N}\left(\frac{\lambda_1}{\mu}\right)^{k-(C-N)} P_0, & \text{for } C-N+1 \le k < C. \end{cases}$$

Blocking probability - new call: This approach is an extension of the MMCC blocking probability method, and is used to calculate the blocking probability for a new call:

$$B_{\text{new}} = \sum_{k=C-N}^{C} P(k)$$

Blocking probability - handover: This approach is an extension of the MMCC blocking probability method, and is used to calculate the blocking probability for a handover call:

$$B_{\text{handover}} = P(C)$$

Aggregated blocking probability: This combines the blocking probability for new calls and handovers, providing a higher weight for handover:

$$B_{\text{aggregated}} = B_{\text{new}} + 10 \times B_{\text{handover}}$$

Server utilisation: the average fraction of time all servers are occupied:

$$U = \frac{L}{C}$$

Average number of customers in system: the mean number of active servers in the system:

$$L = \sum_{k=0}^{C} k \times P(k)$$

2.4 M1M2MCC Actual Formulae

The following are the formulae needed to calculate the above metrics using the simulation data:

Blocking probability - new call:

$$B_{new} = \frac{totallssM1}{totalArrivalsM1}$$

Blocking probability - handover:

$$B_{handover} = \frac{totalLossM1}{totalArrivalsM2}$$

Blocking probability - aggregated:

$$B_{aggregated} = B_{new} + 10 \times B_{handover}$$

Server utilisation:

$$U = \frac{areaServerStatus}{c \times simTime}$$

Average number of customers in the system:

$$L_{actual} = \frac{areaServerStatus}{simTime}$$

3 M/M/C/C Queueing System

The following section describes the design, implementation and testing of the aforementioned M/M/C/C queueing system. The questions detailed in the specification 1.2 and 1.3 will also be covered.

3.1 Design and Code Overview

Before approaching the operation of the queueing simulation, we should first examine the variables and values we use over the course of operation. These are split into distinct categories: state variables, global variables, meta variables and constants.

3.1.1 Constants and meta variables:

A constant is a value that does not change during the simulation. It is predefined and remains fixed, providing a reference or a set parameter that various parts of the simulation use for calculations or to define certain behaviours. Meta variables are variables used outside of a simulation instance, but are used across the course of numerous simulations. Table 1 contains all necessary information regarding the aforementioned variable types.

3.1.2 State variables:

State variables are variables which describe the current state of the system. It captures essential information required to describe the system at any point in the simulation, enabling the tracking of changes over time. Table 2 covers all state variables used in this design, describing type and functionality.

c - This is the number of servers in the system, see constants table 1.

Table 1: Constants and meta variable table

Variable	Type	Description
c	int	The number of servers in the MMCC server system.
SERVICE_AVERAGE	int	This is the reciprocal of the service rate, and dictates the properties of the distribution that produce the service times.
TERMINATION_CRIT	int	This is the no. of arrivals that must occur for the simulation instance to terminate.
ARRIVAL_RATE_MIN	float	This is a meta variable, used over the course of numerous simulations to increment arrivalRate.
ARRIVAL_RATE_MAX	float	This is a meta variable, used over the course of numerous simulations to increment arrivalRate.

Table 2: State variable table

Variable	Type	Description
servers	[boolean]* c	This is a list, where each element represents a server in the system, and their current status (available or not).

3.1.3 Global variables:

Global variables are used to describe the progress of the simulation, and are often used for statistical and temporal measures. They are globally accessible, and guide the simulation to its next operation. Table 3 describes those utilised in this design.

3.1.4 Termination criteria:

Termination criteria in a simulation refer to the predefined conditions or parameters that determine when the simulation should end. These criteria can be based on various factors such as reaching a specific time limit, achieving a desired outcome, the occurrence of a certain event, or meeting specific performance metrics. The simulation stops running once these conditions are satisfied.

Since time is not critical in this experiment, nor is there a particular desired outcome / event, this implementation runs until a certain number of arrivals have occurred. This is constant across all instances of the simulation under one experiment. This criteria is useful when validating and reproducing elements of the model. It also allows performance analysis to be performed effectively.

Our results are produced under two different values for termination criteria. 5,000 arrivals is the first, and acts as a quick heuristic to compare to theoretical results. It can be useful to study the spread of results produced by a smaller simulation too. 500,000 arrivals is the second value implemented, and is used for more accurate, complete simulations. Any results displayed will express the termination value used.

Table 3: Global variable table

Variable	Type	Description
simTime	int	This represents the current simulation time.
timeLast	int	This stores the simTime when the previous event occurred.
nextEventType	char - "a", "d"	Determines the next event type, "a" - arrival, "d" - departure.
timeNextEvent	[int]*c+1	Stores the time of the next event for each server, with an additional element to track the next arrival (final index).
totalArrivals	int	This stores how many arrivals have entered the system.
totalDepartures	int	Stores how many departures occurred.
areaServerStatus	int	This variable represents server utilisation.
arrivalRate	float	Constant throughout an instance of simulation. Represents arrival rate into the queueing system.

3.1.5 Arrival and service time:

Each time an arrival or service is scheduled, a random value for the temporal cost is produced. These values are then recorded appropriately so that the simulation maintains stochasticity, since real life holds uncertainty.

We modelled the service times as an exponential distribution, with a service rate of 0.01. Arrival rate is the independent variable (variable which we study change), although the arrival times will be modelled from Poisson distribution. Arrival rates under consideration will vary between 0.01 and 0.1, as defined by the specification, however, values beyond the specified bounds may be explored too.

3.1.6 Events - Arrival, departure and timing:

There are two main events that can occur during the execution of the simulation; arrival and departure. Another important function is the timing function, which determines the flow if the simulation. These elements of the program will be explored in depth in this section.

Timing:

In the provided simulation code, the timing function plays a crucial role in orchestrating the progression of events. Its primary function is to scan through the timeNextEvent array to pinpoint the next event by identifying the smallest upcoming time value. This step is essential for determining which event is set to occur next in the simulation's timeline. Once this next event is identified, the function proceeds to update the simulation's internal clock, simTime, aligning it with the event's scheduled time. This ensures that the simulation accurately reflects the sequence and timing of events. Additionally, the function returns the index of this impending event, a piece

of information that is used in other parts of the code to discern whether the next event is an arrival or a departure. Through these operations, the timing function effectively manages the chronological order and timing of events, maintaining the integrity and accuracy of the simulation's temporal flow. Code sampled from the implementation is shown in listing 1.

Listing 1: Timing function

```
#Timing function
def timing():
    global simTime
    mIndex = lambda lst: lst.index(min(x for x in lst if x >= 0))
    minIndex = mIndex(timeNextEvent)
    simTime = timeNextEvent[minIndex]
    return minIndex
```

Arrival:

Figure 1 demonstrates the flow of this event. The core of this function involves attempting to allocate an arriving

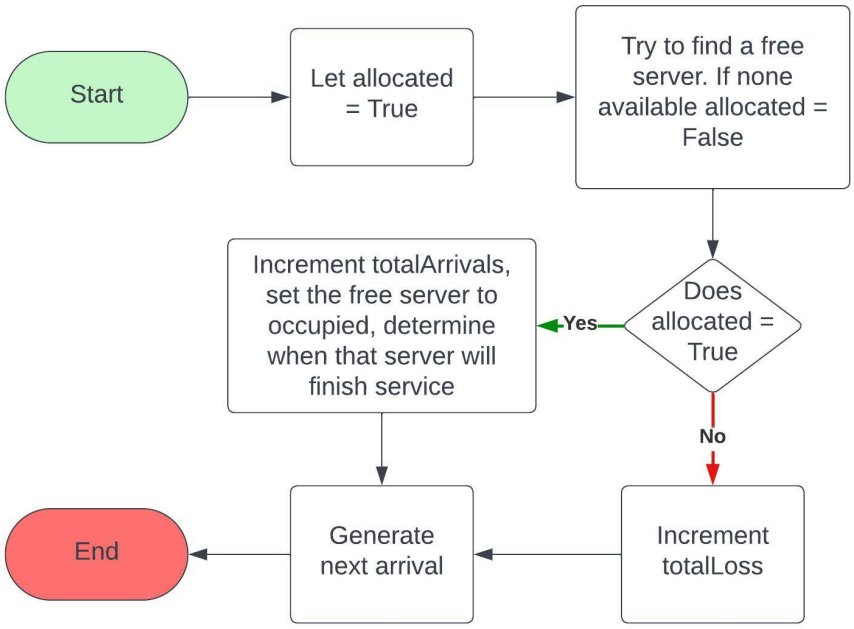

Figure 1: Arrival event flow

customer to an available server. It iterates through the servers list, where each element represents a server's current status (occupied or not). If a free server (indicated by "False") is found, the function allocates the customer to this server, setting its status to "True" and scheduling a departure time for this call based on the service time. If no free server is available, the function increments totalLoss, signifying a lost customer due to all servers being occupied. Subsequently, the function increments totalArrivals to reflect the new arrival and schedules the next arrival event. This procedure effectively simulates the process of customers arriving at a queueing system, being served if possible, or being turned away if all servers are busy. The code snippet in listing 2 shows the exact implementation.

Listing 2: Arrival function

```
#Arrival function
def arrive():
    global totalArrivals, totalLoss, servers, arrivalRate
    allocated = False
    for i in range(len(servers)):
        if servers[i] == False:
            #allocate
            allocated = True
            break
        else:
            #check another server
            pass
    totalArrivals += 1
    if allocated == True:
        servers[i] = True
        timeNextEvent[i] = simTime + ExpService()
    else:
        totalLoss += 1
    timeNextEvent[16] = simTime + ExpArrival(arrivalRate)
```

Some important observations from listing 2 can be made, including the server selection process, being the first available server, and the positioning the totalArrivals increment. It is important that any time an arrival is attempted it is incremented, even in the case where the arrival is lost. It is also worth noting that both servers and timeNextEvent lists use corresponding indices for each server, with the final index of timeNextEvent being reserved for arrivals.

Departure:

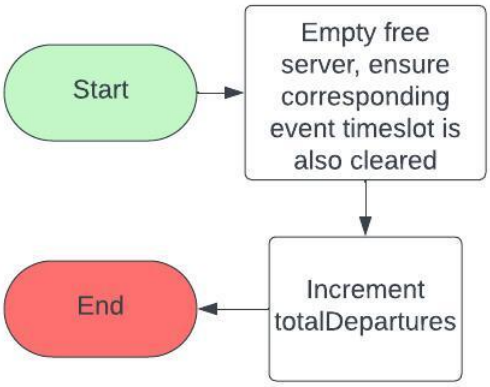

Figure 2: Departure function for MMCC

When invoked, this function primarily interacts with the global variable totalDepartures, which it updates to reflect the number of customers who have completed their service and left the system. The function takes an argument, minIndex, which specifies the index of the server from which a customer is departing. Upon

a customer's departure, two key actions are performed: the server indexed by minIndex in the servers list is marked as available (False), indicating that it is now free to serve another customer, and the timeNextEvent array entry corresponding to this server is reset to -1, signifying that there is no pending event for this server. Consequently, the totalDepartures count is incremented to maintain an accurate record of the total number of departures that have occurred in the simulation up to that point. Overall, the depart function is essential for tracking the availability of servers and updating the simulation's state in response to customer departures.

3.1.7 Main program flow:

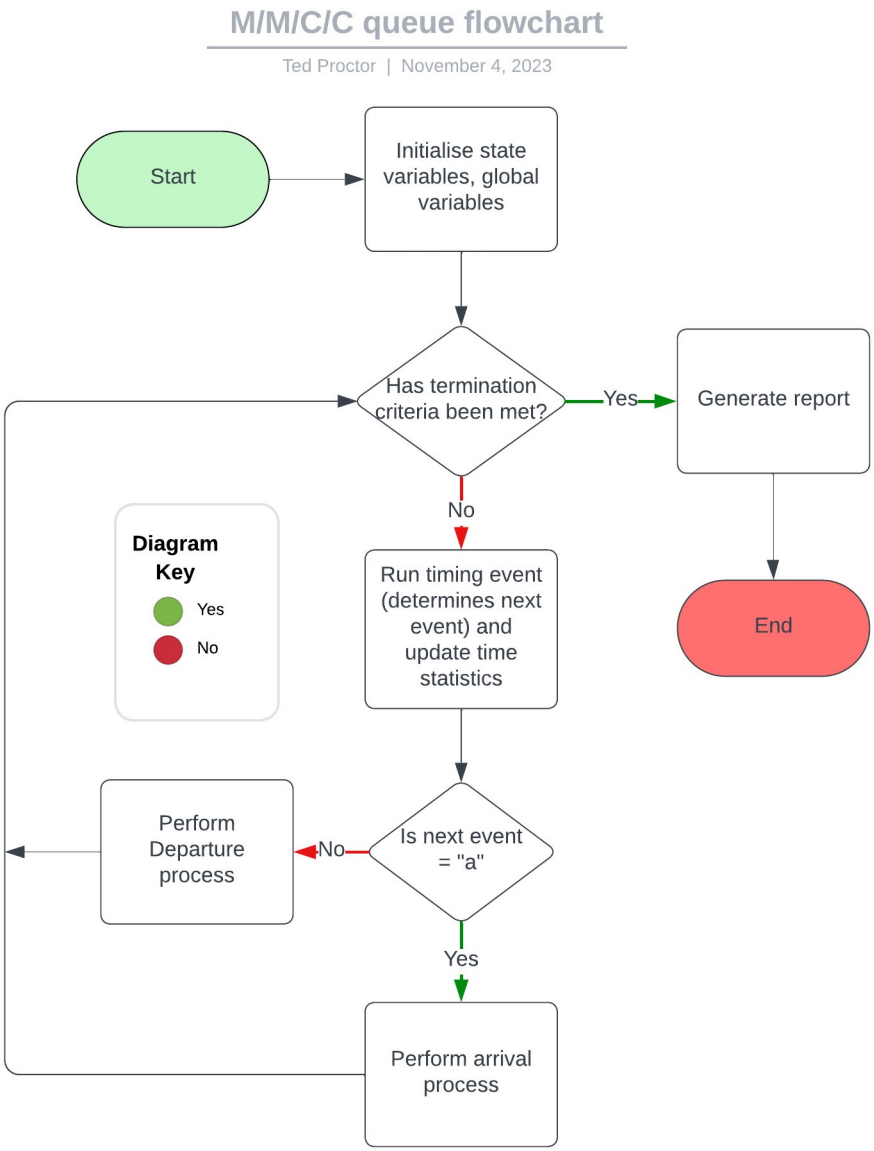

Figure 3: MMCC program flow

The flowchart 3.1.7 outlines the steps for a simulation of an M/M/C/C queueing system, which is a model with 'C' servers where arrivals are turned away if all servers are busy. The process begins by initialising state and global variables, setting up the conditions for the simulation. It then checks whether the termination criteria, such as a time limit or a certain number of customers served, have been met. If not, a timing event is executed to determine the next event and update time statistics. This event decides if the next action is an arrival (denoted by "a") or a departure. In the case of an arrival, the simulation processes the arrival of a new customer or job. If it

is a departure, the process involves removing a customer or job from the system. This loop continues until the termination criteria are satisfied, at which point a report is generated, and the simulation ends. The flowchart uses a green colour to indicate 'Yes' or positive paths and red for 'No' or negative paths in decision-making. The M/M/C/C model assumes no waiting line and memory-less inter-arrival and service times, which means the time between arrivals and the service times are exponentially distributed. Once the simulation has terminated, the program produces a report which details the operation of the simulation, and is used for statistical analysis.

3.2 Implementation Specific Design Choices

For the M/M/C/C queueing system, the decided implementation approach was procedural programming. This was selected due to the simplicity of the queueing system, and because we wanted to test ourselves and implement each of the queueing systems detailed in the rubric under different styles. Subsequently, the M1+M2/M/C/C system will be implemented as object oriented programming. This does bring about the drawback that between each simulation instance, all state variables in the program must be reset.

In an M/M/C/C queuing system with no waiting space, server selection is key to efficiency and implementation complexity.

First Available Server : Chooses the first free server, ensuring a straightforward, deterministic system and potentially O(1) time complexity with the right data structure. This may, however, result in uneven server load.

Random Server Selection : Randomly picks a server until an available one is found, promoting load balance but with a higher time complexity, averaging O(C) when near capacity, with indeterminate worst case complexity. If a more sophisticated approach is taken where a list of available servers is generated *a priori* to selection, the time complexity becomes O(C+1), and the worst case space complexity increase is +O(C).

Last Available Server : The inverse of the first available server, potentially evening out utilisation but possibly under-utilising some servers, with a similar O(1) time complexity given efficient data handling.

While the first and last available server methods are simple and predictable, they might not evenly distribute server load. Random selection balances server load but can be less efficient. The choice hinges on whether load balance or implementation ease is prioritised. In our case, the predictability and simplicity of the first available server made it the preferred choice.

A few termination criterion were considered:

1. X arrivals occur
2. X departures occur
3. Simulation runs for X time - simTime and real world time.
4. X losses occur

When comparing termination criteria for an M/M/C/C queue, "X arrivals occur" is typically more suitable than "X departures occur" or time-based criteria. "X arrivals occur" ensures the simulation runs through enough customer entries to reach a steady state, providing a clear view of the system's handling of incoming traffic and capacity. In contrast, "X departures occur" might not adequately capture the system's response to varying arrival patterns, and since departures are a consequence of arrivals and service times, if the system is initially empty, it will take some time for it to reach a state where departures occur at a steady rate. This delay means that early in the simulation, departures won't accurately reflect the system's capacity to handle arrivals. Time-based

criteria could either cut short before reaching steady state or extend beyond it unnecessarily, and will provide inconsistent results across different machines. Hence, "X arrivals occur" is a preferred choice for accurately assessing system performance in a controlled, predictable manner. Finally, "X arrivals occur" should help in improving the replicability of the simulation.

3.2.1 Statistical measures

In order to correctly record statistics, we need to measure certain elements of the simulation at certain events. The following code snippet describes how the relevant statistical measures are updated:

Listing 3: Update Stats function

```
# update stats (whole simulation)
# This function calculates the areaServerStatus (%active)
# INPUTS - None
# OUTPUTS - None
def updateStats():
    global simTime, timeLast, areaServerStatus
    # calc area, no. of active servers * diff in time
    currentArea = (simTime-timeLast)*servers.count(True)
    # update timeLast
    timeLast = simTime
    areaServerStatus += currentArea
```

This function is used to track the usage of the queueing model, wherein the areaServerStatus variable represents the $\frac{\text{no. of servers in use}}{\text{time}}$.

We must also consider whenever a time state variable changes. This is the variable simTime. It only updates in the timing() function, which is the first step of each iteration.

3.3 Testing

A small test suite was developed in another file to test the key functions in the program. To test the arrival, departure, and timing functions in your M/M/C/C queue simulation, a separate Python script that specifically calls and verifies the functionality of these methods was developed. This test script will simulate a small number of events and check if the functions are updating the system state correctly. They are stored under: tests_for_MMCC.py. Our testing is predominantly performed through the validation phase.

3.4 Validation

Our chosen method of validation is comparing statistical models to analytical models. Validating simulation models against analytical models (2) is effective because it verifies accuracy, identifies errors, and enhances understanding of system behaviour. This comparison ensures the simulation aligns with theoretical expectations, boosting confidence in its results and highlighting areas for improvement. It's a crucial step for establishing the reliability and robustness of the simulation. Our simulation was considered valid if it produced results that aligned with the theoretical results produced by the analytical model. We sampled at a high rate to ensure that the fidelity of our results were sufficient, with the arrivalRate increment being 0.001. We validated our models both visually and using statistical measure mean squared error.

Two of our approaches will be detailed in 3.5.1, however one additional layer of validation was performed, being the average number of customers in the system.

We calculated the theoretical number of customers (thick blue line) in the system using equation 5 and compared it to the recorded number of customers in the system, for a sample of 10 simulations (translucent lines), with their averaged value (thick orange line), as seen in the figures below.

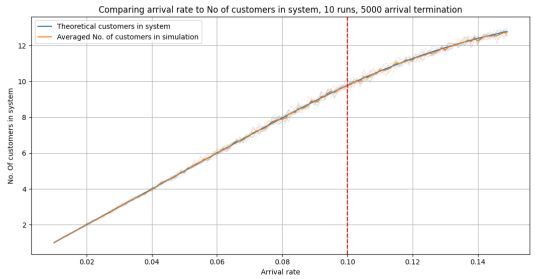

Figure 4: No. of Customers in System, termination criteria = 5000, 10 runs

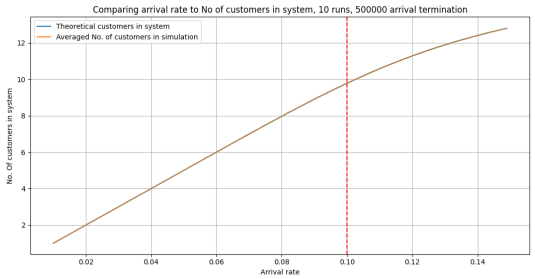

Figure 5: No. of Customers in System, termination criteria = 5000, 50 runs

It is clear the theoretical model closely aligns with the recorded values, as seen by the blue and orange lines being almost identical in fig 3.4, with the exception of some noise in the orange line due to it being taken from simulations with lower termination criteria. This results in less stable results from the simulation. This information is still valuable due to its faster evaluation time. It can also be useful to study the variation from the expected values for shorter simulations.

When the termination criteria increased, the noise the orange line experienced decreased, suggesting an inverse relationship. As shown on fig 3.4, there is little to no deviation from the expected values when the termination criteria has increased. This supports the validity of the model, although other forms of validation should be explored too.

We also examine the similarity between the predicted value and the actual (averaged) value for each of our methods of validation using the root mean squared method. The following tables shows the results of that:

Validation approach	**conditions**	**RMSQ error result**
Blocking Probability	term = 5000, 10 runs	0.001752562383502056
Blocking Probability	term = 500000, 10 runs	0.000824748373002662
Server Utilisation	term = 5000, 10 runs	0.0026219157411826507
Server Utilisation	term = 500000, 10 runs	0.00022499861612933884
No. of Customers in System	term = 5000, 10 runs	0.04792949369785095
No. of Customers in System	term = 500000, 10 runs	0.00413927407526337

Table 4: RMSQ error results for each main configuration

The table overall shows the RMSQ error results for three different validation approaches, each tested over short and long terms (5000 and 500000 respectively) for 10 runs. The pattern indicates that the RMSQ error generally decreases with a longer term, suggesting that the model or system becomes more accurate with more extensive data or over a longer duration.

3.5 Solutions to Exercises

3.5.1 Blocking probability and server utilisation figures:

Blocking Probability: The figures for blocking probability are shown below, with the red dotted demarcation line denoting where the rubric stated the observations should stop (arrivalRate = 0.1), and the theoretical values being calculated using equation 1:

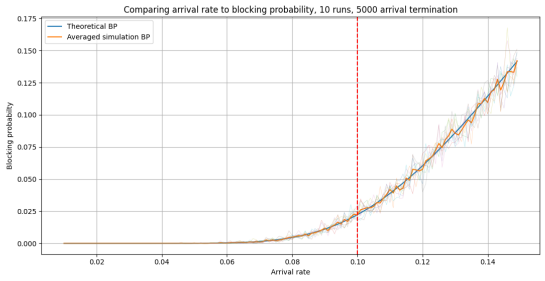

Figure 6: Blocking probability, termination criteria = 5000, 10 runs

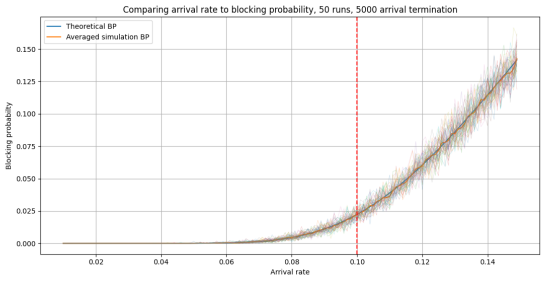

Figure 7: Blocking probability, termination criteria = 5000, 50 runs

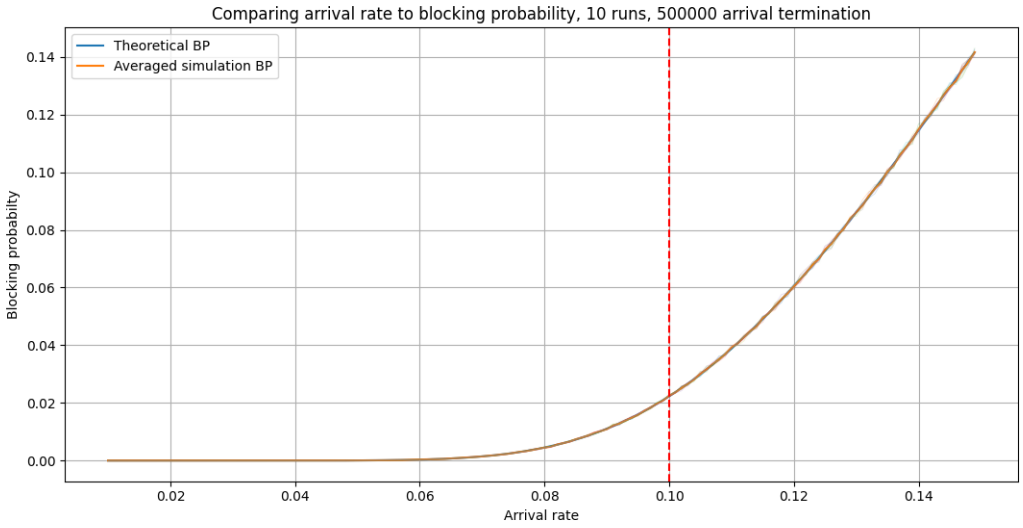

Figure 8: Blocking probability, termination criteria = 500000, 10 runs

As clearly seen, fig 3.5.1 and fig 3.5.1 are remarkably similar, with the one difference being less variation from the predicted values (blue line) in fig 3.5.1, due the the higher number of samples (thinner, translucent lines) used to produce the average (orange line). The expected values match up almost perfectly with what is measured. Both graphs show that the theoretical BP and averaged simulation BP increase as the arrival rate increases.

Fig 8 shows us how the theoretical values compare to the actual values with a significantly larger termination criteria. This allows the model to correct imbalance due to its inherent stochasticity, and should reduce variation, This is the case as the blue and orange lines are almost indistinguishable, let alone the 10 samples (translucent lines) that are used to compose the averaged (orange) line. This indicates that our model is an extremely close match to the theoretical model.

Server Utilisation: Figures for server utilisation are shown below, with the theoretical values being produced by equation 2:

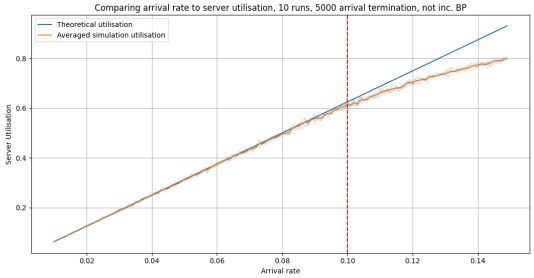

Figure 9: Server Utilisation, termination criteria = 5000, 10 runs, without BP

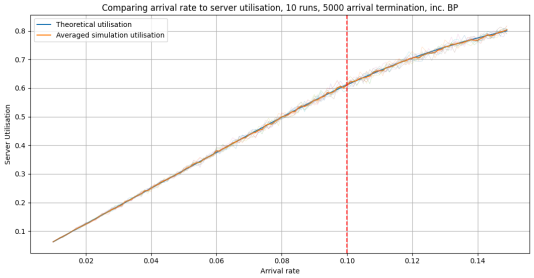

Figure 10: Server Utilisation, termination criteria = 5000, 10 runs, with BP

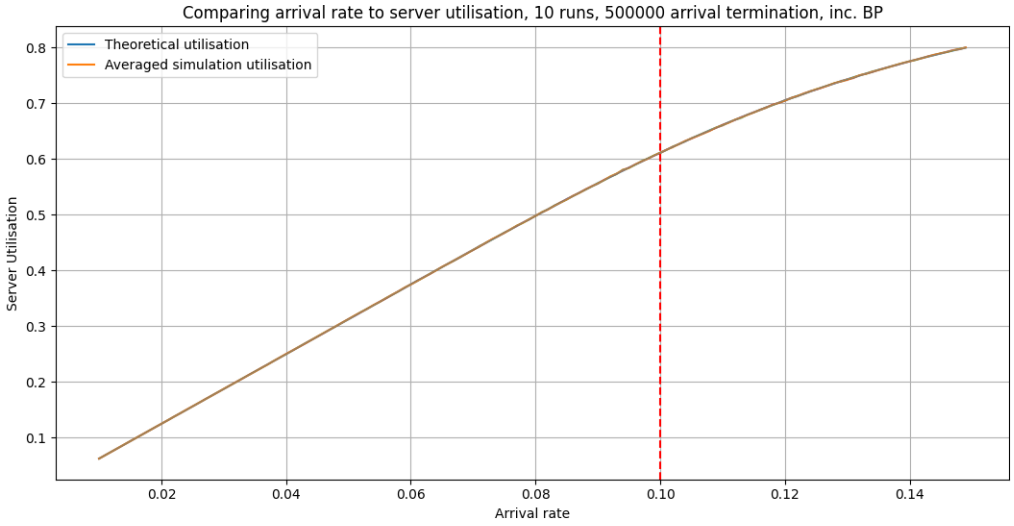

Figure 11: Server Utilisation, termination criteria = 500000, 10 runs. with BP

The approach taken in fig 3.5.1 was the initial approach. It does not consider the affect blocking probability has, and as one can observe, shortly before arrivalRate = 0.1 it begins to diverge from the predicted value. An amendment was made and blocking probability was considered, as shown in fig 3.5.1. With the amended method, the average of our results (orange) matches very closely with the theoretical values (blue), with small variation from the expected results. The samples our average is composed from variate more so, but that is expected for a lower termination value. Fig 11 rectifies this issue, and as seen, there is almost no variation between the expected and actual values, thus validating our model further.

3.5.2 Maximum value for input rates:

Two methods were explored to deduce this value. Both are numerical in nature, one derived from the analytical formula and one from running instances of the simulation and averaging results. The approach is recursive, finding the two boundary floats for arrival rate that make blocking probability < 0.01 and blocking probability > 0.01 respectively, before increasing the accuracy and going in steps one decimal point lower than before. The code is shown below:

Listing 4: Recursive analytical arrivalRate finder

```
# Finds arrivalRate such that BP < 0.01 recursively analytically
# Recursive, calls with decimal points one lower each time
# INPUTS - arrivalRate - recursive
#          step - recursive /10 each time
#          depth - tells when to stop
# OUTPUTS -
def findOptBPAnalytic(arrivalRate, step, depth):
    global c
    # Isolate bounds where it changes using analytical model
    ogArrival = arrivalRate
    BPs = []
    for i in range(10):
        BPs.append([arrivalRate, blockingProbability(arrivalRate, 1/
            SERVICE_AVERAGE, c)])
        arrivalRate += step
    # Locate swap

    index = 0
    for i in range(9):
        if (BPs[i][1] < 0.01 and BPs[i+1][1] > 0.01):
            index = i
            break
    if BPs[9][1] < 0.01:
        index = 9

    if depth == 9:
        return arrivalRate
    else:
        return findOptBPAnalytic(ogArrival + (index/(10**depth)), step/10,
            depth + 1)
```

This approach allows one to determine the decimal accuracy. Due to overflow errors, when the decimal accuracy (depth) is greater than nine, the estimate loses some accuracy. However, an accuracy of nine digits was considered sufficient for the purpose of the exercise. The analytical approach produces a value of 0.08875029 for the arrival rate. This value seems valid, since we can cross-examine our results with fig 8, which indicates that the arrivalRate needed for a blocking probability of 0.01 is approximately 0.09. We also produce results based off of the simulation itself. The approach taken was smilar to previously detailed, but rather than calculating the theoretical blocking probability, we average the blocking probability across 100 simulations, using a termination criteria of 50000 arrivals to ensure higher likelihood of stability in our simulation. The code is as follows:

Listing 5: Recursive real arrivalRate finder

```
def findOptBPReal(arrivalRate, step, depth):
    global c
    # Isolate bounds where it changes using analytical model
    resetStateVars()
    ogArrival = arrivalRate
    BPs = []
    for i in range(10):
        # Generate average
```

```
        averageBP = 0
        for i in range(100):
            repo = main()
            currentBP = repo["totalLoss"] / repo["totalArrivals"]
            averageBP += currentBP
        averageBP /= 100
        BPs.append([arrivalRate, averageBP])
        arrivalRate += step

    # Locate swap
    index = 0
    for i in range(9):
        if (BPs[i][1] < 0.01 and BPs[i+1][1] > 0.01) or (BPs[i][1] < 0.01
            and BPs[i+1] == None):
            index = i
            break

    if depth == 9:
        return arrivalRate
    else:
        return findOptBPReal(ogArrival + (index/(10**depth)), step/10,
            depth+1)
```

Due to the inherent stochasticity of the simulation, this method produced irregular results. To combat this, we ran this method 100 times, and used the set of values to perform some statistical analysis on, such as range and mean. Over the 100 simulation runs, the mean result was $\tilde{0}.0839$. Although this is not identical to the value produced from the analytical model, it is close enough, considering the stochastic nature of the model. Although different, it passes visual inspection of the graph, since the values are similar. The range of arrival rates were as follows: $0 - 0.237$. This is significant variation, which is likely due to the lower termination criteria being employed by this method, to ensure that it can run within reasonable time. This range could be reduced by increasing the termination criteria.

3.6 Warm-up Period

This period refers to the initial phase of a simulation during which the model transitions from its initial state to a steady state. Typically, simulations start with arbitrary initial conditions, such as empty queues or no active users, which are unlikely to represent the system's typical operational state. The warm-up period allows the simulation to reach a point where the effects of these initial conditions have dissipated, and the system behaves more realistically, reflecting its long-term behaviour.

The warm-up period is particularly useful because it helps remove the bias introduced by the initial conditions. This makes the data collected during the simulation more representative of the system's normal operation. For performance measures in queueing systems, like server utilisation, queue lengths, waiting times, and blocking probabilities, accuracy is higher when the system is in a steady state. The warm-up period is essential for the system to reach this state.

A pertinent consideration for determining our warm-up period is its variability. Each run will produce variable results due to the in-built stochasticity, therefore it would be wise to average our results over a high number of runs (50). We also want to ensure that the warm-up period holds true across all of our test cases. We know that as arrival rate increases, so does the server utilisation, and that the initial case covered (where

arrivalRate = 0.01, and c = 16), the rate of arrival and service are equivalent, leading to a stable simulation. To combat this issue, we will deduce the warm-up period suitable for the highest studied arrivalRate, being 0.1. This will cover all cases, since the expected warm-up period for cases with smaller arrival rate should be shorter. We will not calculate the warm-up period for each arrivalRate since each step of this model is very cheap to evaluate, and the additional security is worth the cost. Welch's method builds on this approach with the inclusion of windows, where data is averaged from a number of samples to improve the stability.

Resultantly, our approach is defined as:

1. Initial conditions: Maximum value for arrival rate (0.1), all other values remain the same.
2. for some sample rate X, find the blocking probability and server utilisation averages and record.
3. increment X by 10, repeat.
4. average values over windows of length 100 steps
5. isolate sample Y where blocking probability or server utilisation plateaus, use the larger of the two since stricter ruling will provide more stable simulation.
6. Increase by some flat rate to negate stochasticity.

The following figures (12, 13) display our results from applying Welch's method.

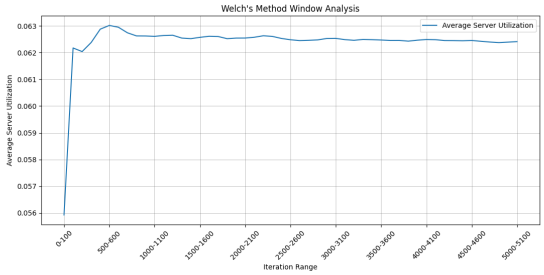

Figure 12: Welch's method, first run

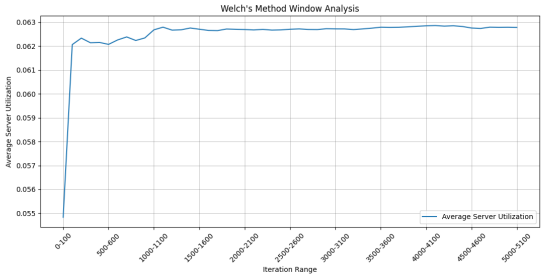

Figure 13: Welch's method, second run

We can see great similarity between the two figures, with very similar server utilisation rates being produced. In fig 12, the server utilisation has stabilised at around the 700-800 arrivals mark. This differs from fig 13 which stabilises at around 1100-1200 arrivals. To clarify this variation we averaged the results over 500 simulations, producing figure 14:

Over a greater number of simulations, the moving point (window) stabilised far earlier, at $\tilde{2}$00-300 arrivals. This is not consistent with our previous observations, and despite implying that the average number of arrivals until the simulation stabilises is lower, it does not consider worst case scenarios. Due to this, we decided to select 1200 arrivals as the warm-up period, with an additional 100 arrivals added to account for stochasticity.

We regenerated our existing results considering the warm-up period, as shown in figure 15:

As one can see, implementing the warm-up period in fig 15 has made no difference to the accuracy of our model. This is likely due to the simplicity of the model. Simple models or those that reach a steady state quickly might not be significantly affected by a warm-up period. If the transient behaviour of the system has a negligible impact on the overall performance metrics, the warm-up period might not be necessary. Alternatively, if the initial state of the simulation is set close to the steady-state conditions or if the initial conditions are not far from typical system behaviour, the warm-up period might have a minimal effect on the model's output quality. Regardless of the reason. The addition of a warm-up period does not benefit this simulation.

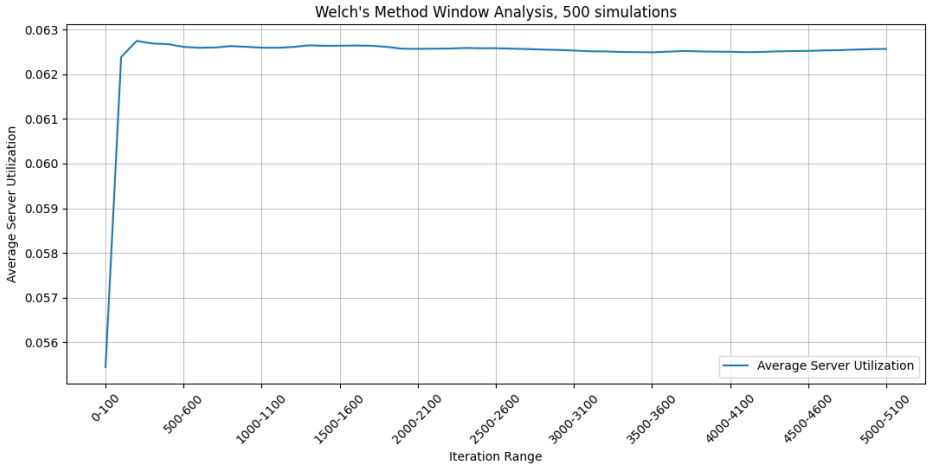

Figure 14: Welch's method, third run, 500 simulations averaged

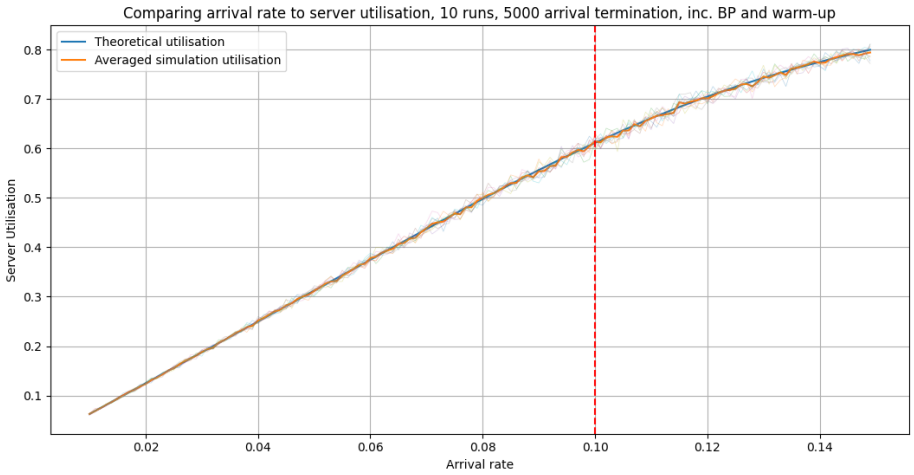

Figure 15: Server utilisation with warm-up

4 M1+M2/M/C/C Queueing System

4.1 Design and Code Overview

The following subsections cover the implementation of the M1+M2/M/C/C.

4.1.1 Constants and meta variables:

This implementation includes all the previous constants and meta variables detailed in 1, with some additions that are detailed in the table below. Furthermore, ARRIVAL_RATE_MIN and ARRIVAL_RATE_MAX are adapted for the M1 arrival rate (new calls), and new ones are made for handover rate. Table 5 details these adaptions.

Table 5: Constants and meta variable table M1+M2MCC

Variable	Type	Description
ARRIVALM1_RATE_MIN	float	This is a meta variable, used over the course of numerous simulations to increment new call arrivalRate.
ARRIVALM1_RATE_MAX	float	This is a meta variable, used over the course of numerous simulations to increment new call arrivalRate.
ARRIVALM2_RATE_MIN	float	This is a meta variable, used over the course of numerous simulations to increment handover arrivalRate.
ARRIVALM2_RATE_MAX	float	This is a meta variable, used over the course of numerous simulations to increment handover arrivalRate.

4.1.2 State variables:

These are identical to the state variables for MMCC queues, detailed in Table 2.

4.1.3 Global variables:

The global variables used in the M1+M2MCC queue are the same as they are in MMCC, however some additional ones must be implemented or altered to accomodate for two arrival rates. They are detailed in Table 6

4.1.4 Arrival and service time:

Similar to MMCC queues, we modelled the service times as an exponential distribution, with a service rate of 0.01. We have two arrival rates (one for new calls and one for handover), both independent variables, with values generated from a Poisson distribution. Arrival rates under consideration will vary, and the model will be validated under different conditions for arrival rates, where one is considered constant as the other variates.

Table 6: Global variable table M1+M2MCC

Variable	Type	Description
arrivalM1	float	Represents arrival rate for new calls.
arrivalM2	float	Represents arrival rate for handover.
totalArrivalsM1	int	Number of new calls into simulation.
totalArrivalsM2	int	Number of handover calls into the simulation.
totalLossM1	int	Number of new calls rejected by simulation.
totalLossM2	int	Number of handover calls rejected by the simulation.
timeNextEvent	[int]*c+2	Stores the time of next event, with additional two elements, one for next new call (penultimate index), one for next handover call (ultimate index).

4.1.5 Events - Arrivals, departure and timing:

This approach extends the MMCC queue. There are three events that can now occur, departure, new call arrival, and handover arrival. The departure method is the same.

Timing: This function operates identically to the timing method described for MMCC. The only difference is that the approach is modified to accommodate for OOP, and the list that the function operates on is one element longer, to account for two arrival types.

Listing 6: Python code for timing function

```
#timing function
#find the next event
def timing(self):
    #find next event, min in next event times
    mIndex = lambda lst: lst.index(min(x for x in lst if x >= 0))
    minIndex = mIndex(self.timeNextEvent)
    #update system clock
    self.simTime = self.timeNextEvent[minIndex]
    #update sim times inside
    return minIndex
```

New call arrival: It begins by identifying free servers and increments the count of M1 arrivals. If free servers exceed a certain threshold, the function allocates the first available server and schedules its next service event. Otherwise, it logs the arrival as rejected and increments the loss count. Finally, the function sets the timing for the next arrival based on the M1 arrival rate. The following listing demonstrates the code.

Listing 7: Python code for handling arrival in M1

```
def arrivalM1(self):
    #var declaration
    freeServers = []
    #find out how many servers are free
    for i in range (len(self.servers)):
        if self.servers[i] == False:
            freeServers.append(i)

    #increment arrivals for M1
    self.totalArrivalsM1 += 1

    #if there are free servers considering the free spaces for handover
        calls
    if len(freeServers) > self.threshold:
        #freeServers[0] is the index of the first free server
        self.servers[freeServers[0]] = True
        self.timeNextEvent[freeServers[0]] = self.simTime + self.
            expService()
    #reject arrival
    else:
        self.totalLossM1 += 1

    #generate next arrival
    self.timeNextEvent[self.c] = self.simTime + self.expArrival(self.
        arrivalRateM1)
```

The following flowchart (16) is another representation of the programmatic flow:

Handover arrival: It first initialises a variable 'allocated' as 'False' to track if a server allocation occurs. The function then iterates through the list of servers to find an available one, marking 'allocated' as 'True' if a free server is found. Regardless of server availability, the total number of M2 arrivals is incremented. If a server is available, the function sets this server's state to busy and schedules its next service event. If no server is available, the function increments the total M2 losses count. Finally, the function calculates and sets the time for the next M2 arrival based on the specified M2 arrival rate.

Listing 8: Python code for handling M2 arrivals

```
#arrival M2 - handover
#allocates a handover call to any free server, if not will increment
    totalLosses. Generates new handover arrival
#on completion
def arrivalM2(self):
    #vars declaration
    allocated = False
    #allocate arrival to free state var, give service time
    for i in range (len(self.servers)):
        if self.servers[i] == False:
            allocated = True
            break
        else:
```

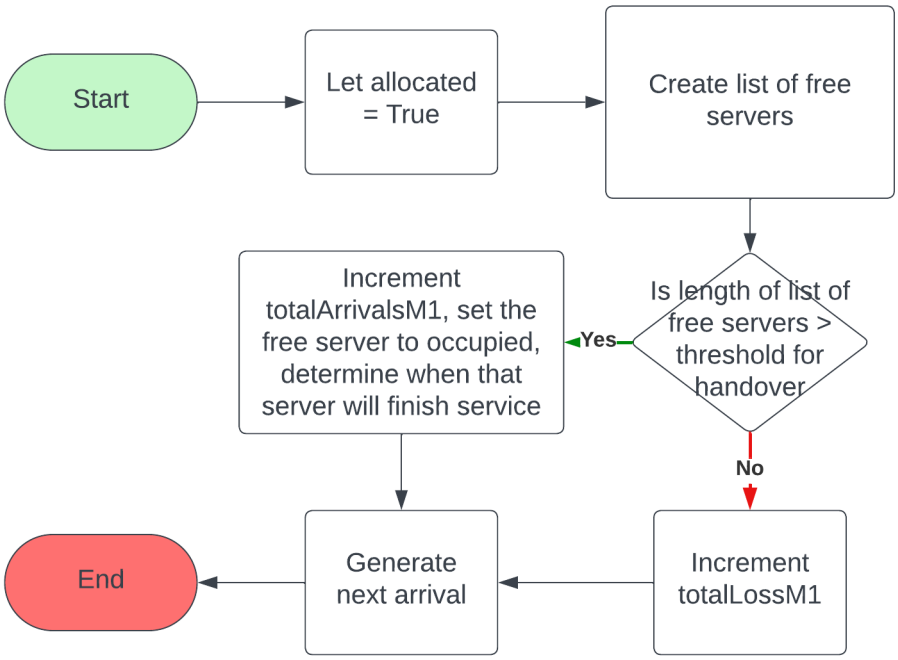

Figure 16: Arrival flowchart for new arrival

```
            pass
    #alter state vars accordingly
    self.totalArrivalsM2 += 1
    if allocated == True:
        self.servers[i] = True
        self.timeNextEvent[i] = self.simTime + self.expService()
    else:
        self.totalLossM2 += 1
    #generate next arrival for handover
    self.timeNextEvent[self.c+1] = self.simTime + self.expArrival(self.
       arrivalRateM2)
```

Although it functions very similarly to the arrival function for MMCC queues, it is important to note that totalArrivalsM2 and totalLossM2 are incremented, rather than the generic versions. The flowchart 17 demonstrates this.

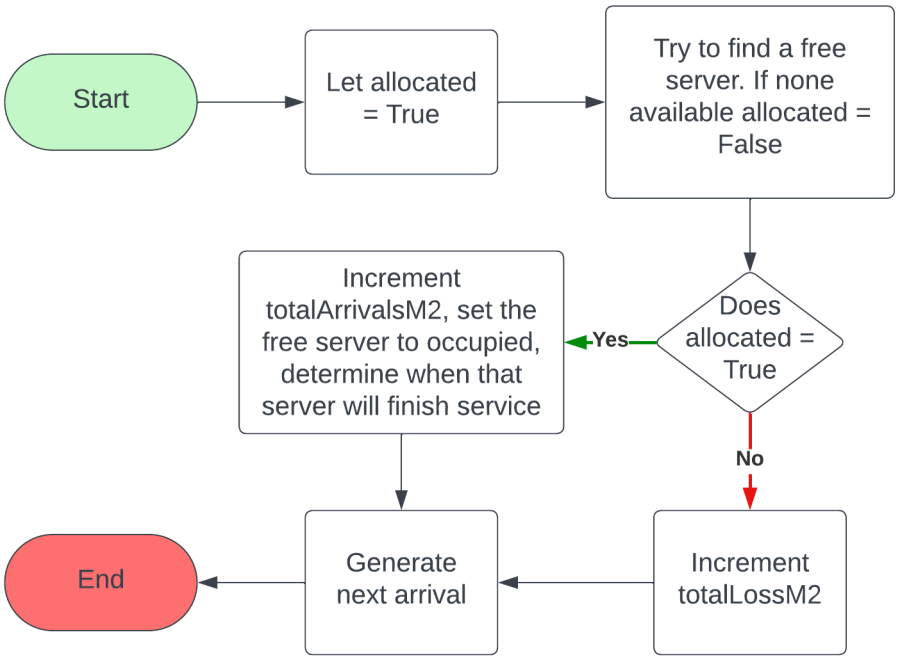

Figure 17: Arrival flowchart for handover

4.1.6 Main program flow:

The runSimulation (see fig 18) function orchestrates a simulation process within a queueing system. Initially, it sets the next event times for two types of arrivals, M1 and M2, based on their respective arrival rates. The simulation then enters a loop, which continues until a specified termination condition is met. Within each iteration of the loop, the function determines the next event to process using the 'timing' method. After updating statistical data, it handles the event based on its type: if it's an M2 arrival (handover call), it calls 'arrivalM2'; if it's an M1 arrival (new call), it calls 'arrivalM1'; and if it's a departure, it calls 'departure' for the corresponding server. The loop checks after each iteration whether the combined total of M1 and M2 arrivals has reached a pre-defined termination threshold. Once this condition is satisfied, the simulation ends, and the function returns a report generated by report().

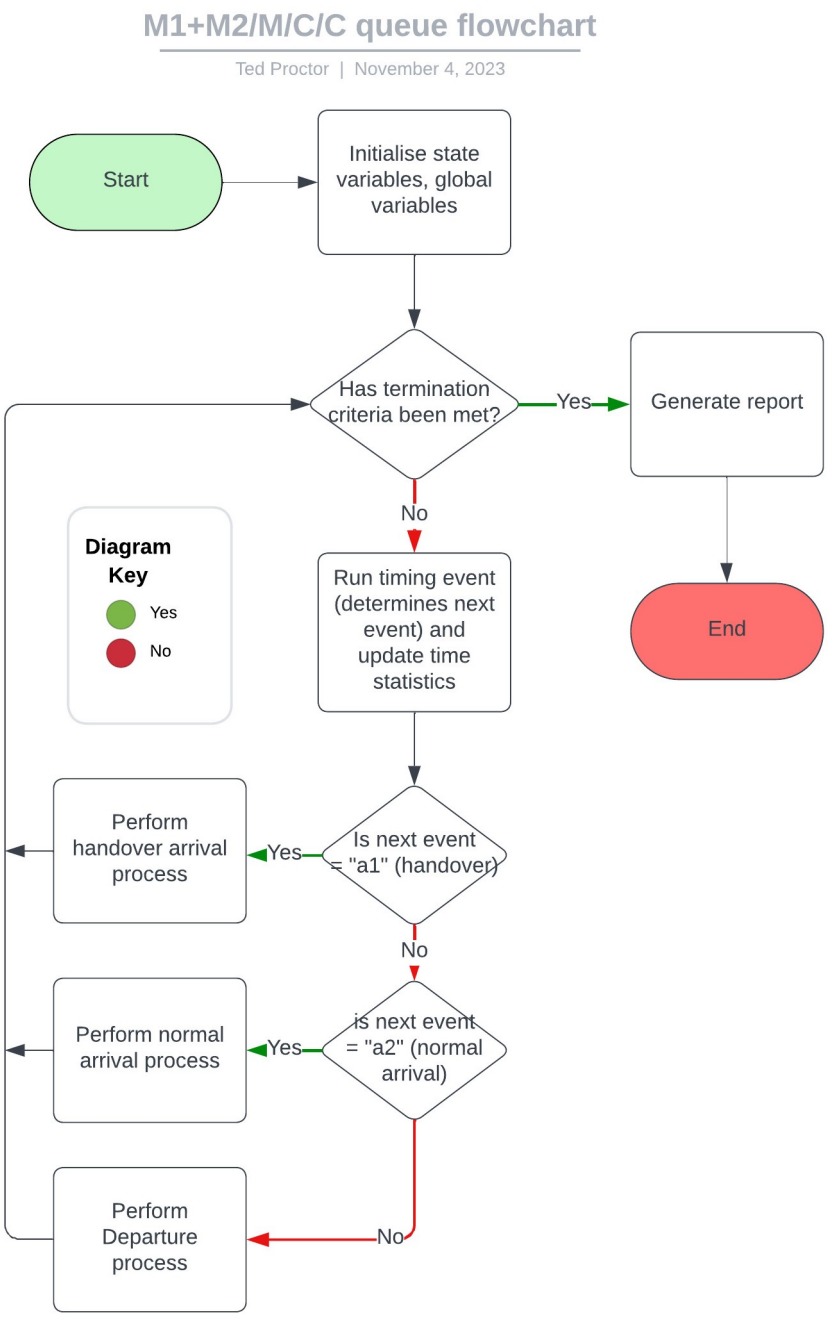

Figure 18: Flowchart for M1+M2/M/C/C queueing system

4.2 Implementation specific design choices

We employ the same server selection method as the MMCC queue, being 'first available server', and the same termination criteria choice of 'X arrivals occur' This is due to their success in the first implementation. The only significant difference is the programming approach. In the MMCC queue the decided approach was procedural programming, whereas for this queue and object-oriented-programming (OOP) approach was taken. This was to improve our understanding of the problem by undertaking two different programmatic approaches, and to test our implementation skills. One advantage of the OOP approach, which was uncovered *a posteriori*, was the exclusion of the resetAllVariables() function, which was necessary to call between individual runs of the simulation for the procedural approach. This slowed down the execution of compound simulations, since there would be some up-time between each run to allow the simulation to reset.

4.2.1 Statistical measures:

The approach was identical to the one covered in MMCC queues. The following code demonstrates our operations:

Listing 9: Python code for updating statistics

```
def updateStats(self):
    #for each server, if server is active increment server util based on
       simTime and timeLast
    for i in range(len(self.servers)):
        if self.servers[i] == True:
            self.areaServerStatus[i] += self.simTime - self.timeLast
    #update timeLast to be simTime
    self.timeLast = self.simTime
```

4.3 Testing

Rather than developing a test suite, testing was performed manually on the three events. The report() function was utilised, and by examining the difference between two the provided reports (which produce a comprehensive report on all variables within the system), before and after an event was called, we could visually verify the correctness of all events. This proved to be less time-consuming that the test suite developed for the MMCC queue, with the same necessary coverage.

4.4 Validation

In order to validate our models when we have two variables, for each validation method we had to set one of our variables (handover arrival rate or new call arrival rate) to some arbitrary constant (in our case we set the non-variable arrival rate to 0.1). To allow for faster evaluation, we used a lower termination criteria of 10,000 arrivals, ensuring it was still large enough for the simulation to enter a steady state. The service time mean persisted from the MMCC queue model, remaining at 100 seconds. We incremented the arrival rate under consideration by 0.005, with a lower limit of 0.01, and an upper limit of 0.5. This ensures good coverage of values and allows a better evaluation of our validation. Finally, each arrival rate was evaluated 10 times by the simulation model, with individual runs being demonstrated by the thinner grey lines, and the average results being shown by the blue line. The following set of graphs (Fig 19, 20, and 21) show the variation of new call arrival rates to various validation metrics: The figures displays three separate graphs, each correlating new call

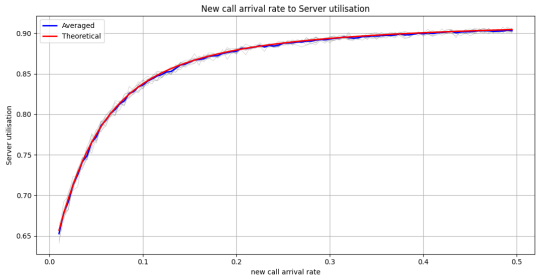

Figure 19: New call arrival rate to server utilisation

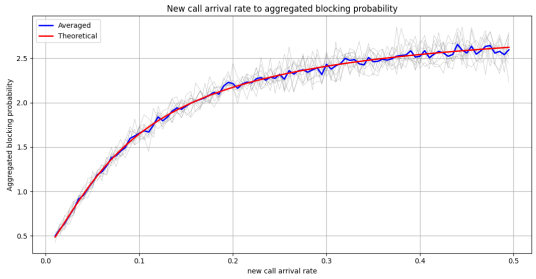

Figure 20: New call arrival rate to aggregated blocking probability

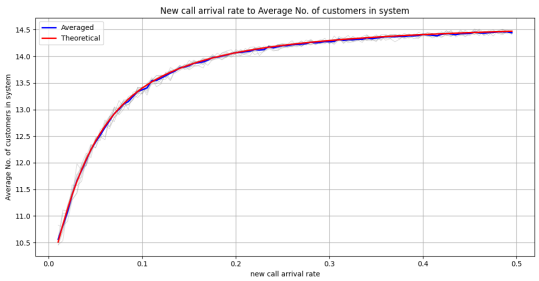

Figure 21: New call arrival rate to average No. of customers in the system

arrival rates with distinct system performance metrics, with two lines representing averaged and theoretical results.

In Figure 19], titled "New call arrival rate to server utilisation," both the averaged and theoretical lines increase sharply with the arrival rate before reaching a plateau, indicating a point of maximum utilisation capacity for the servers. The lines are nearly indistinguishable from each other, implying that the averaged empirical results closely match the theoretical predictions throughout the range of arrival rates.

Figure 20, labelled "New call arrival rate to aggregated blocking probability," shows an uptrend where the aggregated blocking probability ascends with the arrival rate. The averaged and theoretical lines run closely parallel, with the averaged line showing a slight divergence and more variability as the arrival rate increases. However, the overall closeness of the two lines suggests that the theoretical model provides a solid approximation of the real-world data.

In Figure 21, "New call arrival rate to average No. of customers in the system," the trend is similar to the first graph: both lines rise with increasing arrival rates, reaching a level-off point. Here again, the averaged and theoretical lines track each other closely, with minimal deviation, underscoring the theoretical model's effectiveness in estimating the average number of customers in the system across varying arrival rates.

The proximity of the averaged (empirical) and theoretical lines across all three graphs indicates that the theoretical model is highly representative of the actual system behaviour over the observed range of new call arrival rates. The consistency across all metrics highlights the model's robustness.

The following set of graphs (Fig 22, 23, and 24) show the variation of handover arrival rates to various validation metrics:

These figures show very similar results to the first set, with all empirical results closely aligning with the theoretical results. This indicates full validity of our model since both variables have been validated across a range of inputs. To reinforce this we can apply the root mean squared test to all to data points to ensure a measure of their similarity. Table **??** shows the averaged RSMQ error across all values for the variable.

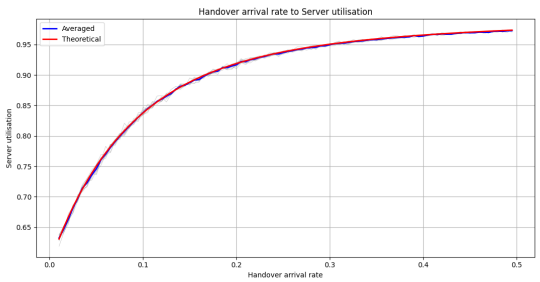

Figure 22: Handover arrival rate to server utilisation

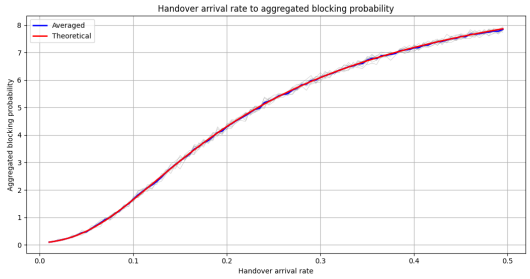

Figure 23: Handover arrival rate to aggregated blocking probability

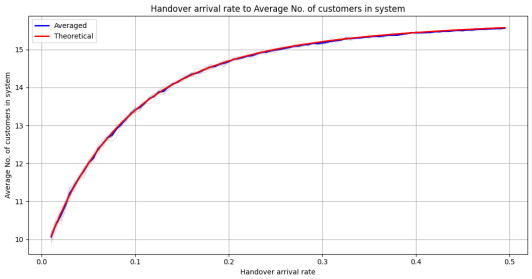

Figure 24: Handover arrival rate to average No. of customers in the system

Validation figure	RMSQ error
Fig 23	0.024718264003500103
Fig 22	0.001109068635059519
Fig 24	0.017745098160952304
Fig 20	0.029127398297014336
Fig 19	0.001224351648385828
Fig 21	0.019589626374173247

Table 7: Caption

Thus compounds the evidence that our model is valid, as evident through the consistently low RMSQ error values.

4.5 Solutions to Exercises

4.5.1 Aggregated blocking probability:

In Exercise 2.2, we aim to analyse and determine the maximum handover call arrival rate (M2) in a wireless cell network characterised by an M1+M2MCC queuing system, while ensuring that the Aggregated Blocking Probability (ABP) remains below a threshold of 0.02. This exercise builds upon the established simulation environment from Exercise 2.1, where the system parameters include 16 servers (C=16) and a threshold for new call acceptance set at 2. The scenario under investigation involves a fixed new call arrival rate (M1) of 0.1 calls per second, and the call duration's are exponentially distributed with an average of 100 seconds. Through iterative simulation runs, each employing a gradually increasing handover rate, we seek to pinpoint the maximum rate at which handover calls can be accommodated without surpassing the specified ABP limit. This approach provides valuable insights into the system's capacity and robustness in handling varying levels of call and handover traffic.

To start we can adapt our validation graph 23, such that the new call arrival rate is set to 0.01, rather than 0.1, as the specification demands. Using this graph, we can deduce approximately the handover rate for the set conditions to be met. By zooming into a specific part of our the results (Fig 25) we can see that the green line (where ABP = 0.02) intersects our theoretical results at $\tilde{0}.0592$, suggesting that our calculated value should be approximately there.

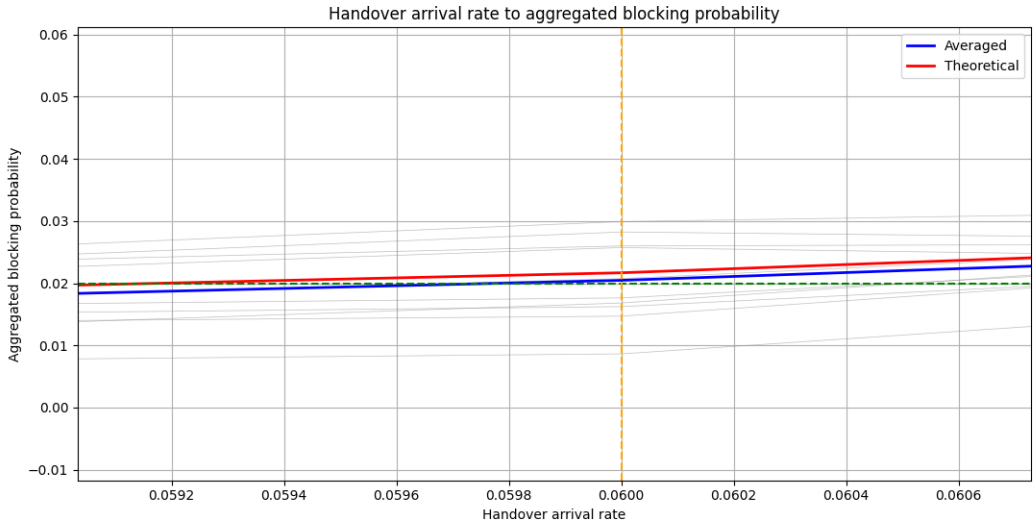

Figure 25: Handover values, given new call arrival rate of 0.01

To verify this information, a binary search was employed to discover its exact value, using the analytical model for aggregated blocking probability. An upper bound of 0.06 was provided, and a lower bound of 0. The following listing demonstrates the code:

Listing 10: Binary search for handover

```
def findHandover():
    # Parameters
    c = 16
    threshold = 2
    serviceAverage = 100
    arrivalsForTerm = 10000  # Adjust as needed
    arrivalRateM1 = 0.1

    # Binary search bounds
    low_bound = -0.5
    high_bound = 0.06  # Adjust based on expected maximum
    precision = 0.000001  # Define the precision of your search

    # Binary search loop
    while high_bound - low_bound > precision:
        print("doing")
        arrivalRateM2 = (low_bound + high_bound) / 2
        #simulation = QueueSimulation(c, threshold, arrivalRateM1,
            arrivalRateM2, serviceAverage, arrivalsForTerm)
        #report = simulation.runSimulation()
        # Calculate probabilities
```

```
        #CBP = report['totalLossM1'] / report['totalArrivalsM1']
        #HFP = report['totalLossM2'] / report['totalArrivalsM2']
        #ABP = CBP + 10 * HFP

        analytical = AnalyticalModels(c=16, threshold=2, arrivalM1 =
            arrivalRateM1, arrivalM2 = arrivalRateM2, serviceRate = 1/
            serviceAverage)
        ABP = (analytical.newCallBP())+10*(analytical.handoverBP())
        # Adjust bounds based on ABP
        if ABP < 0.02:
            low_bound = arrivalRateM2
        else:
            high_bound = arrivalRateM2

    # Result
    max_handover_rate = low_bound
    print(f"The maximum handover rate with ABP < 0.02 is approximately: {
        max_handover_rate}")
```

On execution, the program returned the lower bound as the maximum probability. This occurred over a range of sample values, and on closer inspection of the provided aggregate blocking probability formula, a deductive process was employed:

$$\text{Call Blocking Probability (CBP)} = \frac{\text{total\_new\_call\_loss}}{\text{total\_new\_call\_arrivals}}; \tag{6}$$

$$\text{Handover Failure Probability (HFP)} = \frac{\text{total\_handover\_loss}}{\text{total\_handover\_arrivals}}; \tag{7}$$

$$\text{Aggregated Blocking Probability (ABP)} = \text{CBP} + 10 \times \text{HFP} \tag{8}$$

Given this, we know that:

$$\text{ABP} < 0.02 \tag{9}$$

$$\text{CBP} + 10 \times \text{HFP} < 0.02 \tag{10}$$

We evaluate CBP with $\lambda_1 = 0.1$ and $\lambda_2 = 0$ (since that is the minimum possible value) and find that:

$$\text{CBP} = 0.05681914338652086 \tag{11}$$

Now we can evaluate HFP with the minimised values and find:

$$\text{HFP} = 0 \tag{12}$$

$$0.05681914338652086 + 10 \times 0 > 0.02 \tag{13}$$

This contradicts the earlier statement that ABP < 0.02,

Therefore, it is impossible, since negative arrival rates cannot occur.

(14)

Thus concluding that maximum value for the handover rate so that the Aggregated Blocking Probability (ABP) < 0.02 does not exist.

4.5.2 Maximum value for input rates:

This task involves determining the maximum new call arrival rate (measured in calls per second) for a telecommunications system under the condition that the handover call rate is fixed at 0.03 handovers per second. The goal is to find the highest rate at which new calls can arrive without causing the ABP to exceed a threshold of 0.02. The task requires careful analysis or simulation of the queuing system to assess how different rates of new call arrivals affect the ABP. We will employ a similar approach as before, but first we must verify that a solution exists.

$$\text{Call Blocking Probability (CBP)} = \frac{\text{total\_new\_call\_loss}}{\text{total\_new\_call\_arrivals}}; \tag{15}$$

$$\text{Handover Failure Probability (HFP)} = \frac{\text{total\_handover\_loss}}{\text{total\_handover\_arrivals}}; \tag{16}$$

$$\text{Aggregated Blocking Probability (ABP)} = \text{CBP} + 10 \times \text{HFP} \tag{17}$$

Given this, we know that:

$$\text{ABP} < 0.02 \tag{18}$$

$$\text{CBP} + 10 \times \text{HFP} < 0.02 \tag{19}$$

We evaluate HFP with $\lambda_1 = 0$ and $\lambda_2 = 0.03$ (since that is the minimum possible value) and find that:

$$\text{CBP} = 1.024323285425509e-07 \tag{20}$$

Now we can evaluate CBP with the minimised values and find:

$$\text{CBP} = 3.3802668419041797e-06 \tag{21}$$

$$3.3802668419041797e-06 + 10 \times (1.024323285425509e-07) < 0.02 \tag{22}$$

This aligns with the earlier statement that ABP < 0.02,

Therefore, a solution is viable

(23)

Since we know a solution exists, we first correspond with the graph (Fig. 26) given some conditions to decide on bounds for our binary search.

From this we can deduce it is likely that the new call arrival rate value that satisfies our will be between the bounds of 0.04 and 0.05. This must be taken with a grain of salt, since our visual analysis of ex 2.2 was somewhat misleading. We apply these bounds to the binary search algorithm detailed in the previous section. The binary search produced the following result: "The maximum new call arrival rate with ABP < 0.02 is approximately: 0.045985412597656254". We can validate this result by plugging the value into our aggregate blocking formula (Fig 27):

This is extremely close to the constraint ABP < 0.02. although the precision could theoretically be increased, one must consider the effects of floating-point error and accept some error margin. Six decimal points of accuracy was deemed sufficient for this exercise.

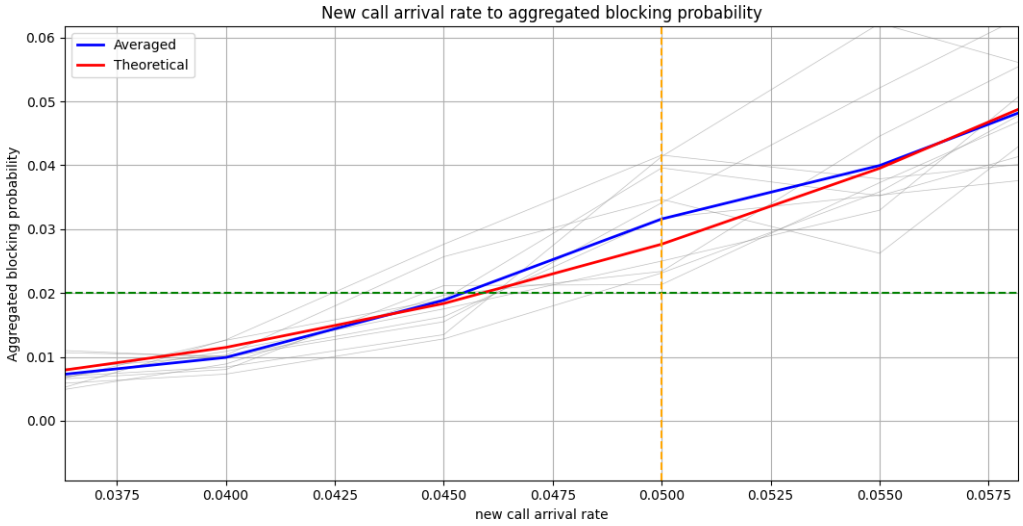

Figure 26: new call values, given a handover arrival rate of 0.03

```
aModel = AnalyticalModels(c=16, threshold=2, arrivalM1 = 0.045985412597656254, arrivalM2 = 0.03, serviceRate = 1/100)
print(aModel.newCallBP() + (10*aModel.handoverBP()))
```

Figure 27: Code to validate result

```
0.019999459139157613
```

Figure 28: Result of code from Fig 27

5 Program Execution

As requested, a section is dedicated to program execution. A single run of both the MMCC and the M1M2MCC queue will be documented. Composite runs will not be documented as their outputs have been refined into the figures witnessed through the course of this report.

5.1 MMCC running

:

```
PS C:\Users\TheodoreProctor\OneDrive - Define Research & Insight\Documents\Masters\Computer Modelling and Simulation (ECMM424_A_1_202324)\CW>
```

Figure 29: Before execution

```
618    main()
```

Figure 30: Execution command

```
roctor/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/TheodoreProctor/OneDrive - Define Research & Insight/Documents/Masters/Computer Modelling and
Simulation (ECMM424_A_1_202324)/CW/MMCC_queue.py"
{'noOfServers': 16, 'arrivalRate': 0.01, 'simEndTime': 998361.1912680387, 'totalArrivals': 10000, 'totalDepartures': 9997, 'totalLoss': 0, 'areaServerStatus': 100
5186.652639555, 'areaServerStatusProp': 0.0629272915849002}
PS C:\Users\TheodoreProctor\OneDrive - Define Research & Insight\Documents\Masters\Computer Modelling and Simulation (ECMM424_A_1_202324)\CW>
```

Figure 31: After Execution

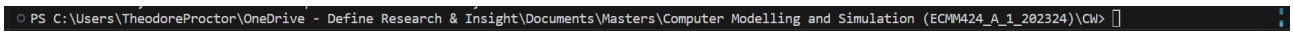

Figure 32: Before execution

```
721   qSim = QueueSimulation(c=16, threshold=2, arrivalRateM1=0.01, arrivalRateM2 = 0.1, serviceAverage=1/100, arrivalsForTerm=10000)
722   print(qSim.report())
```

Figure 33: Execution command

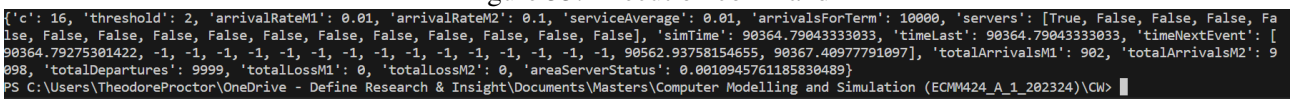

Figure 34: After execution

5.2 M1M2MCC running

6 Improvements and further research

Improving an M1M2MCC queue can focus on several key areas, especially in terms of server selection and managing server degradation.

For server selection, adopting a dynamic approach can significantly enhance performance. Traditional methods like round-robin or random selection might not be efficient for complex queue systems. Instead, consider implementing a load-balancing algorithm that accounts for the current load and potential processing time of each server. This can be based on machine learning algorithms that predict server availability and workload, thereby optimising the distribution of tasks. Another approach is priority-based server selection, where tasks are assigned to servers based on priority levels of the tasks, ensuring high-priority tasks are processed faster.

Additionally, addressing server degradation is crucial for maintaining the system's efficiency and longevity. Implementing predictive maintenance using machine learning can forecast potential server failures or performance issues, allowing for proactive maintenance and reducing downtime. Regularly updating the server selection algorithm based on historical performance data also ensures that degraded servers are less likely to be chosen, maintaining the overall efficiency of the system.

These improvements would require a physical manifestation of the simulation to work at maximum operational capacity, and many elements of the model would need to be refined in order for these methods to hold value.

By focusing on these areas, you can significantly enhance the performance and reliability of an M1M2MCC queue system, or any queueing system, ensuring it can handle complex scenarios with higher efficiency.

7 Conclusions

This report rigorously examined and simulated two distinct queuing models: M/M/C/C and M1+M2/M/C/C, both key to understanding cellular network dynamics. The creation and implementation of these simulations yielded deep insights into their operational mechanisms.

The study delved into the M/M/C/C model, a representation of a single-cell cellular network, focusing on its handling of call traffic through arrivals and departures. Its Markovian process framework ensures an accurate depiction of cellular network traffic.

The M1+M2/M/C/C model introduced a higher level of complexity by integrating a priority system for new and handover calls, using a threshold parameter. This model realistically emulates scenarios in cellular networks

where existing calls take precedence over new ones, thus mirroring the networks' operational practices. The adoption of Object-Oriented programming significantly improved the code's readability and management.

Both systems underwent rigorous testing and validation to confirm their reliability and accuracy. The alignment of simulation outcomes with theoretical expectations across various parameters underscored the models' robustness. These exercises highlighted practical applications of the models, delineating the operational limits of these implementations.

To summarise, the coursework effectively demonstrated the simulation of critical queuing systems in cellular networks, underscoring the value of merging theoretical analysis with practical simulation exercises.