

ECMM424 - Computer Modelling and Simulation

Coursework: $M/M/C/C$ and $M1+M2/M/C/C$ Queue Simulation

ALEX RUNDLE, University of Exeter, UK

I certify that all material in this report which is not my own work has been identified and correctly credited:

Signed: Alex Rundle

1 EXERCISE 1: M/M/C/C QUEUE

Only snippets of code are seen in this report, including only those snippets that are important in the context of this report. The whole code can be seen in `MMC.py`.

1.1 Program Structure and Execution

For the M/M/C/C queue, I chose to implement the simulation in Python 3.9 as a Class with the following methods:

- `main()`: Runs the decision loop for the entire simulation
- `timing()`: Controls the timing of the simulation and determines the next event
- `update_time_avg_stats()`: Updates the time average of Server Status
- `arrive()`: Signifies an arrival in the system
- `depart()`: Signifies a departure from a given server in the system
- `report()`: Calculates the total server utilization of the system
- `expon(mean)`: Generates a number from an exponential distribution, using the given mean
- `block_prob()`: Calculates the blocking probability for a given number of servers

The time of the next event is stored in an array, `self.time_next_event`, and takes the value of `inf` for every index. The time for the next arrival event is held in index 0, with each index after representing a departure from a server e.g. index 1 represents the time of next departure from server 1. The value `inf` is used to represent the fact that no event has yet been scheduled e.g. if `self.time_next_event[3] = inf`, server 3 is not expected to have a departure.

Further, the status of each server is kept in the array `self.server_status`, where a value of zero means the server at that index is idle and a value of one means the server is busy.

Although this queue is modelling an M/M/C/C queue, in which there is no queue space, the queue limit is still included during initialisation and is set to zero; the logic for adding and removing calls from the queue remains so that this class could be used for regular M/M/C simulation.

For both this simulation and the M1+M2/C/C simulation seen in Section 2, I chose a 'Selection in Order' selection rule, in which the first idle server found in order (by iterating over servers 1 through 15) is selected. Although this does not aid with equalizing server utilization, the mean service rate and number of servers are both high enough that the first server should not be overloaded and over-utilized compared to the other servers later in order.

To calculate the utilization and blocking probabilities for the graphs seen in Section 1.2, the following code is used:

```

1  util_arr = []
2  block_arr = []
3
4  for i in range(1, 11, 1):
5      q = MMCQueue(
6          q_limit=0,
7          mean_arrive=i/100,
8          mean_service=100,
```

```

9         cust_req=100,
10         num_servers=16,
11     )
12     q.main()
13     util_arr.append(q.total_server_utilization)
14     E = q.mean_arrive / q.mean_service
15     block_arr.append(q.block_prob(E, q.num_servers))

```

The resulting arrays, `util_arr`, `block_arr`, is then be copied into a MATLAB file to create the graphs. The code to create the blocking graph, Figure 1, can be seen here:

```

1  arrival = [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1];
2  blocking = [4.77899940855074e-78, 3.131651871541877e-73, 2.056791142623393e-70,
3  2.0519489397040275e-68, 7.28925810402878e-67, 1.3475343222878723e-65, 1.5872489202512062e-64,
4  1.3442274585884605e-63, 8.848500753677113e-63, 4.774700243997285e-62];
5
6  block = plot(arrival, blocking);
7  x1 = xlabel('Mean Arrival Rate, $\lambda$');
8  set(x1, 'Interpreter', 'latex');
9  y1 = ylabel('Blocking Probability, $P_{\{c\}}$');
10 set(y1, 'Interpreter', 'latex');
11 t1 = title({'Plot of Blocking Probability against Mean Arrival Rate'; []});
12 set(t1, 'Interpreter', 'latex');
13 set(gca, 'XLim', [0.01 0.1])
14
15 block.Color = 'r';
16 block.LineWidth = 1;
17 block.Marker = 'x';
18 block.MarkerEdgeColor = 'black';
19 saveas(gcf, 'block_fig.png')

```

The code to generate the Server Utilization graph, Figure 2, is near identical in structure as the above code and was omitted.

1.2 Blocking Probability and Server Utilization Figures

For this exercise, the simulation was run once for each arrival rate (values in the range [0.01, 0.1], incremented by 0.01). The blocking probability was calculated using the Erlang B formula [1], [2]:

$$B(E, m) = \frac{\frac{E^m}{m!}}{\sum_{k=0}^m \frac{E^k}{k!}} \quad (1)$$

where $E = \frac{\lambda}{\mu}$. The function `block_prob()` calculates the blocking probability via a recursive calculation of the inverse of the blocking probability; this is done to ensure numerical stability.

```

1  def block_prob(self, E: float, m: int) -> float:
2      """The blocking probability for a given number of servers
3
4      Uses the Erlang B formula, running a recursive calculation.
5
6      Parameters
7      -----
8      E : float
9          Offered load i.e. mean_arrive / mean_service
10     m : int
11         Number of Servers
12
13     Returns
14     -----
15     float
16         A float representing the blocking probability
17     """
18     inv_b = 1.0
19     for j in range(1, m+1):
20         inv_b = 1.0 + inv_b * j / E
21     return 1.0 / inv_b

```

The above function can be explained mathematically with Equations 2 and 3.

$$\frac{1}{B(E, 0)} = 1 \quad (2)$$

$$\frac{1}{B(E, j)} = 1 + \frac{j}{E} \frac{1}{B(E, j-1)} \quad \forall j = 1, 2, \dots, m \quad (3)$$

For all blocking probability values, the mean service time μ is constant at $\mu = 100$. As can be seen in Figure 1, the blocking probability is very small for the arrival rates chosen, having a minimum value of 4.78×10^{-78} at mean arrival $\lambda = 0.01$.

Although Figure 1 suggests that the blocking probability does not change much between mean arrivals 0.01 through 0.07, it actually increases rapidly through 14 orders of magnitude; this occurs from 4.78×10^{-78} at $\lambda = 0.01$ to 1.59×10^{-64} , at $\lambda = 0.1$. This does suggest that blocking probability is massively affected by the Mean Arrival Rate. Further, this graph suggests that for the currently chosen Mean Service time ($\mu = 100$ seconds), the current mean arrival rates are more than sufficient to run the simulation with a very low blocking probability.

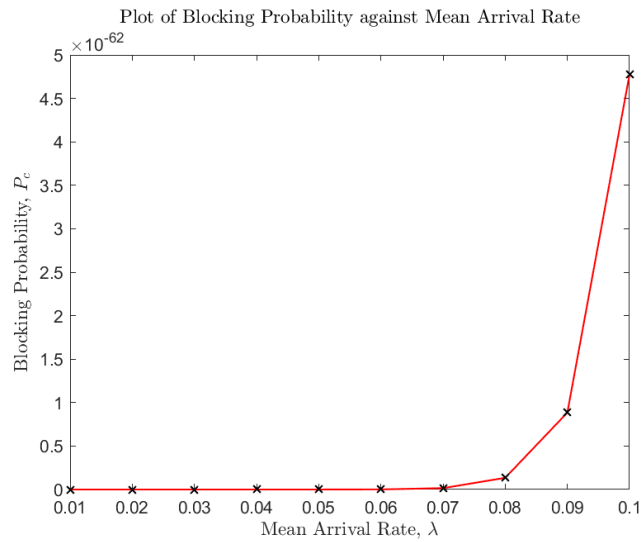


Fig. 1. Blocking Probability increases as Mean Arrival Rate increases

One can see from Figure 2 that Total Server Utilization varies wildly between arrival rates $\lambda = 0.01$ and $\lambda = 0.1$. There are minimum values seen at $\lambda = 0.04$ (where total server utilization = 0.694) and $\lambda = 0.07$ (where total server utilization = 0.761) suggesting that these values see the highest amount of idle time for all servers. This further suggests that these arrival rate values are the best to choose combined with the number of servers, $c = 16$, and for this specific mean service time, $\mu = 100$.

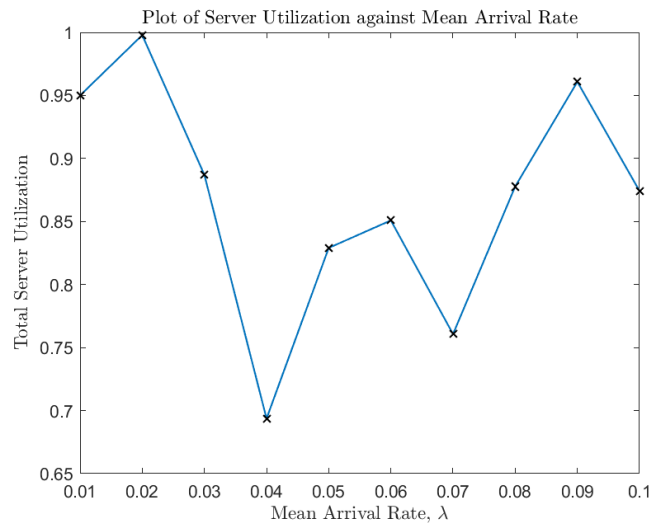


Fig. 2. Server Utilization Tends to be erratic, but tends to decrease to minimums at $\lambda = 0.04$ and $\lambda = 0.07$

1.3 Finding Maximum Arrival Rate for a low Blocking Probability

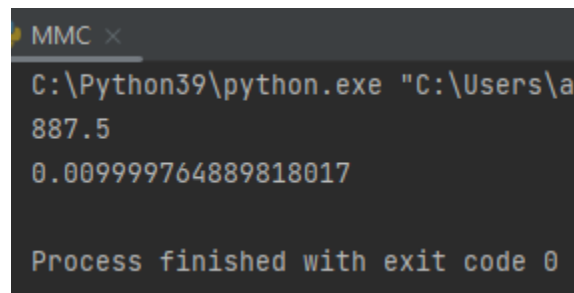
To find the maximum arrival rate that satisfies $B(E, m) < 0.01$, the function `find_max_block_prob()` is used. This function instantiates an M/M/C/C queue similar to the previous sections and calculates the blocking probability for said queue. The queue itself is instantiated repeatedly with increasing values of mean arrival rate λ , incremented each iteration by 0.01. To avoid the inaccuracies of float representations (where some numbers cannot be properly represented as floats during calculations) the calculations are done with integers, with the subsequent results being divided by 100.

```

1 def find_max_block_prob():
2     """Finds the maximum value for arrival rate
3
4     Finds the max arrival rate value such that blocking probability is < 0.01
5
6     Returns
7     -----
8     max_value : float
9         Float for the maximum arrival rate value
10    """
11    mean_arrival = 1
12    last_mean = 0.0
13    done = False
14
15    while not done:
16        q = MMCQueue(
17            q_limit=0,
18            mean_arrive=mean_arrival/100,
19            mean_service=100,
20            cust_req=100,
21            num_servers=16)
22        E = q.mean_arrive / q.mean_service
23        blocking_prob = q.block_prob(E, q.num_servers)
24
25        if blocking_prob >= 0.01:
26            break
27        else:
28            last_mean = mean_arrival
29            mean_arrival += 1
30
31    return last_mean/100

```

For each mean arrival rate, the blocking probability is calculated and checked if larger than 0.01; if the blocking probability is larger than 0.01 for the current mean, the mean from the previous iteration is returned. Running this function yields the mean arrival rate of $\lambda = 887.5$, as seen in Figure 3. Further proof can be seen underneath the found arrival rate, being the blocking probability of an M/M/C/C queue instantiated with the mean arrival rate $\lambda = 887.5$, clearly being less than 0.01. A more precise mean arrival rate might be found by increasing the precision of the increments, for example by using an increment of 0.001 instead of 0.01. However, for the case of this report I have chosen to limit the calculations to 1 decimal place; in experiments with finer precision, the mean arrival rate's second and third decimal place were zero and hence omitted.



```
MMC x
C:\Python39\python.exe "C:\Users\ad...
887.5
0.009999764889818017

Process finished with exit code 0
```

Fig. 3. Execution of the function `find_max_block_prob()`, in which the highest mean arrival rate is found

The code that was used to find the value provided in Figure 3 can be seen below:

```
1  if __name__ == '__main__':
2      max_arrival = find_max_block_prob()
3      print(max_arrival)
4
5      q = MMCQueue(
6          q_limit=0,
7          mean_arrive=max_arrival,
8          mean_service=100,
9          cust_req=100,
10         num_servers=16,
11     )
12     q.main()
13     E = q.mean_arrive / q.mean_service
14     print(q.block_prob(E, q.num_servers))
```

2 EXERCISE 2: PRIORITY M1 + M2/M/C/C QUEUE WITH HANDOVER

Only snippets of code are seen in this report, including only those snippets that are important in the context of this report. The whole code can be seen in `M1M2MC.py`.

2.1 Program Code and Execution

For the M1+M2/M/C/C queue, I chose to implement the simulation in a very similar way to the implementation of Section 1. The main decision loop remains mainly the same, with two types of arrival event now instead of one, representing the new call arrivals and handover arrivals. Further, the structure of the original arrays tracking key properties (e.g. `self.num_events`, `self.time_next_event`) have had to be extended by 2 elements to accommodate the two arrival types. Where index 0 represented an arrival and indices 1 through 15 represented departures from servers, in this case the indices 0 and 1 represent a new call arrival and handover call arrival respectively. The methods for the class `M1M2MCQueue` can be seen below:

- `main()`: Runs the decision loop for the entire simulation
- `timing()`: Controls the timing of the simulation and determines the next event
- `update_time_avg_stats()`: Updates the time average of Server Status
- `arrive(type)`: Signifies an arrival in the system, where type is either a new call or handover
- `depart()`: Signifies a departure from a given server in the system
- `report()`: Calculates the total server utilization of the system
- `expon(mean)`: Generates a number from an exponential distribution, using the given mean
- `agg_block_prob()`: Calculates the aggregated blocking probability
- `total_idle()`: Finds the total number of currently idle servers

Again, the 'Select in Order' server selection rule is used for the same reasons as Section 1.

2.2 Finding Maximum Handover Arrival Rate for low ABP

For this section, the function `find_max_handover_rate()` was used. This function, as seen below, runs a simulation of the M1 + M2/M/C/C Queue for varying handover arrival rates, to find the maximum value that the Handover Arrival Rate can take such that the Aggregated Blocking Probability < 0.02.

```

1  def find_max_handover_arrival() -> float:
2      """Finds the maximum handover arrival rate such that ABP < 0.02.
3
4      Returns
5      -----
6      float
7          The maximum handover arrival rate such that ABP < 0.02
8      """
9      mean_arrival = 1 # Mean starts at 1 to avoid float rep errors
10     last_mean = 0.0
11     done = False
12
13     while not done:
14         q = M1M2MCQueue( # Instantiate new queue
15             q_limit=0,
16             mean_m1_arrive=0.1,
```



```

17     mean_m2_arrive=mean_arrival/10,
18     mean_service=100,
19     cust_req=100,
20     num_servers=16,
21     threshold=2,
22 )
23 q.main() # Run simulation
24 print(mean_arrival/10)
25 abp = q.agg_block_prob() # Calculate ABP for sim
26
27 if abp < 0.02: # If the blocking prob is less than 0.02, stop searching
28     done = False
29 else:
30     last_mean = mean_arrival
31     mean_arrival += 1
32
33 return last_mean/10 # Divide int arrival to provide the original float

```

The arrival rate starts at $\lambda_2 = 0.1$ and is incremented by 0.1 each iteration, checking whether the current ABP is less than 0.02.

Aggregated Blocking Probability is calculated as:

$$CBP = \frac{total_new_call_loss}{total_new_call_arrivals} \quad (4)$$

$$HFP = \frac{total_handover_loss}{total_handover_arrivals} \quad (5)$$

$$ABP = CBP + 10 \times HFP \quad (6)$$

However, due to the fact that the handover calls are accepted as long as there is at least one idle server and that new calls are only accepted as long as the number of idle servers reaches a certain threshold, the change in handover arrival rate cannot cause ABP to achieve a value less than 0.02. As the arrival rate increases, the HFP seen in Equation 5 decreases; the CBP seen in Equation 4, however, does not decrease. The handover calls are served with priority and thus new calls are either blocked more often or with the same probability as previous simulations. With the CBP not decreasing, this causes the ABP itself to fluctuate around a minimum value close to the CBP. As the HFP is nearing a value of zero, the CBP acts as the minimum value that the ABP can reach. Thus, the ABP cannot reach values less than 0.02 without change to the New Call Arrival rate as well.

As can be seen in Figure 4, the ABP for the mean handover arrival rates from $\lambda_2 = 132.7$ through $\lambda_2 = 134.1$ are shown, with the ABP fluctuating about $ABP \approx 0.98$. This suggests that the value for CBP may be around 0.98. Further, this function showed the ABP value of $ABP \approx 10.83$ at the first iteration, at mean handover arrival rate $\lambda_2 = 0.1$; the ABP values decreased to around 0.98 at arrival rate $\lambda_2 = 26.3$.

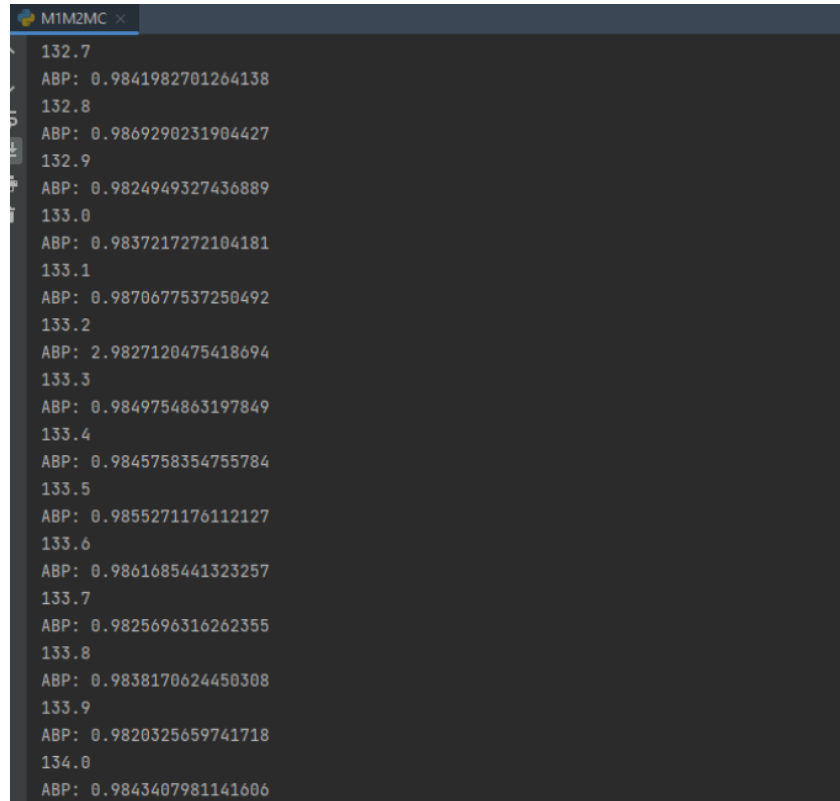


Fig. 4. Log of Aggregated Blocking Probability for increasing Handover Call Arrival Rate (seen above the ABP values)

2.3 Finding Maximum New Call Arrival Rate for low ABP

Similar to Section 2.2, the function `find_max_new_arrival()` runs the simulation multiple times with varying new call arrival rates, λ_1 , starting from $\lambda_1 = 0.1$ and increasing by 0.1 each time. This function can be seen below:

```

1 def find_max_new_arrival() -> float:
2     """Finds the maximum new call arrival rate such that ABP < 0.02.
3
4     Returns
5     -----
6     float
7         The maximum new call arrival rate such that ABP < 0.02
8     """
9     mean_arrival = 1 # Mean starts at 1 to avoid float rep errors
10    last_mean = 0.0

```

```

11     done = False
12
13     while not done:
14         q = M1M2MCQueue( # Instantiate new queue
15             q_limit=0,
16             mean_m1_arrive=mean_arrival/10,
17             mean_m2_arrive=0.03,
18             mean_service=100,
19             cust_req=100,
20             num_servers=16,
21             threshold=2,
22         )
23         q.main() # Run simulation
24         print(mean_arrival/10)
25         abp = q.agg_block_prob() # Calculate ABP for sim
26
27         if abp < 0.02: # If the blocking prob is less than 0.02, stop searching
28             done = False
29         else:
30             last_mean = mean_arrival
31             mean_arrival += 1
32
33     return last_mean/10 # Divide int arrival to provide the original float

```

Aggregated Blocking Probability is again calculated using Equations 4, 5 and 6. As with Section 2.2, the change in New Call Arrival rate only affects the value of CBP; this means that the value for HFP does not change when the new call arrival rate is changed, meaning the ABP value still fluctuates about a minimum value that is close to that of the HFP. Seen in Figure 5, each line of execution shows the current mean arrival rate being ran, followed by the Aggregated Blocking Probability calculated for said mean. This screenshot is taken over the same range of mean handover arrival rates as Figure 4, $\lambda_2 = 132.7$ to $\lambda_2 = 134.0$, and it can be seen that the ABP fluctuates about the value $ABP \approx 10.94$. Seeming as the Handover Fail Probability has ten times the significance as Call Blocking Probability in the ABP calculation (equation 6), it tends to suggest that when HFP is still very high (within the range $0.9 \leq HFP \leq 1.0$), ABP would be slightly larger than $ABP \approx 10$.

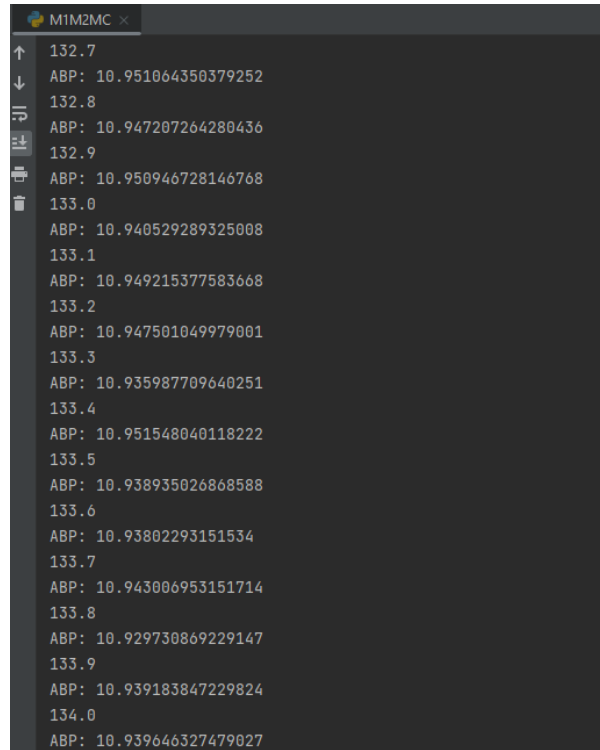


Fig. 5. Log of Aggregated Blocking Probability for increasing New Call Arrival Rate (seen above the ABP values)

REFERENCES

- [1] G. Zeng, "Two common properties of the erlang-b function, erlang-c function, and engset blocking function," *Mathematical and computer modelling*, vol. 37, no. 12-13, pp. 1287–1296, 2003.
- [2] E. Messerli, "Bstj brief: Proof of a convexity property of the erlang b formula," *The Bell System Technical Journal*, vol. 51, no. 4, pp. 951–953, 1972.