ECMM424: Modelling and Simulation. M/M/C/C and M1-M2/M/C/C Queue Modelling

Paul Workman*

Foreword

This report is created as part of the submission for ECMM424: Computer Modelling and Simulation's Continuous Assessment. It explains the methods used to implement M/M/C/C and M1-M2/M/C/C simulations, and the methods used to test and validate the implementations. It also includes screenshots of the program at run-time. This report includes a 36m-20s demonstration and code review which also details the program's run-time and decisions on how to implement the queues. It is strongly recommended, that, to understand the code fully, and to generally reduce the amount of time to go through the review, you watch the demonstration first. The video report can be found here: https://youtu.be/u9rqRRSv7vE. NOTE! The video presentation includes a bug where the ABP calculation is not correctly divided when returning. If you have watched the video before reading this paper, page 14 onward contains novel information not included in the demonstration.

I certify that all material in this dissertation which is not my own work has been identified.

Signature: Paul Jonathan Workman

* pw466@exeter.ac.uk

1. Code overview

1.1. M/M/C/C Queue Simulation

The M/M/C/C Queue implementation relies on 3 different classes, which all have a series of methods. I decided to take an object-oriented approach as the capabilities of customers and servers indicated that each should be subjected to its separate methods. This function also allowed attributes to be set for each instance of customers and servers allowing for a quick and easy translation of all information to CSV and string formats. The implementation and the methods for each class are described below.

1. Customer

```python
class Customer:
    """Class representing each a customer to be served
    by a service.

    @attributes:
        id -- the customers unique ID, ordered by birth, earlier<later
        birth_time -- the global time of customer creation and queue entering
        death_time -- the global time a customer is finished being served
        service_time -- the time spent being served
        served_by -- the server ID of the instance serving the customer
        rejected -- a bool containing whether the customer was rejected
    """
    id : int = 0

    birth_time : int = 0
    death_time : int = 0

    service_time : int | None = 0
    served_by : int | None = None

    rejected : bool = False
```

Fig. 1. Customer Class Attributes.

The customer class attributes contain information regarding the ID, birth time, death time, and service time of the customer. This allows for the program on output to easily list the information in all of the customers created during the simulation. Additionally, the class contains information about the server it is served by using the ID of the server and also contains a boolean to describe whether the customer was rejected upon creation.

```python
def __init__(self, time : int, given_id : int) -> None:
    self.birth_time = time
    self.id = given_id

def reject(self, time : int) -> None:
    """Used to set internal attributes to represent a customer has been rejected

    @parameters
        time - the global time of rejection
    """
    self.rejected = True
    self.service_time = None
    self.served_by = None
    self.death_time = time
    logging.info("Rejecting and killing customer %s", repr(self))
```

Fig. 2. Customer class initialisation and rejection.

The customer class attributes contain information regarding the ID, birth time, death time, and service time of the customer. This allows for the program on output to easily list the information in all of the customers created during the simulation. Additionally, the class contains information about the server it is served by using the ID of the server and also contains a boolean to describe whether the customer was rejected upon creation.

```
def serve(self, server : 'UniversalServer', time: int) -> None:
    """Used to set interal attributes to represent currently being served

    @parameters:
        server -- the server object which is serving the customer
        time -- the global time a cutomer beings to be served
    """
    self.served_by = server.id
    self.service_time = time
    logging.info("Serving customer %s with server: %s", repr(self), repr(server))

def kill(self, time : int) -> None:
    """Used to set internal parameters to represent a cutomer has
    been completed an no longer needs to be considered

    @parameters:
        time -- the global time a customer is finished with
    """
    self.death_time = time
    logging.info("Killing customer %s", repr(self))
```

Fig. 3. Customer class service and kill.

For the serve and kill methods of the customer class upon the serve method being utilized the served by attribute is updated to be the server ID and the service time is set by a given time. The kill method is used when a service has been fulfilled and a server lets the customer go. Upon calling the kill function the death time is set to the given time. This does not delete the customer instance.

```
def __str__(self) -> str:
    return f"Customer ID: {self.id} " \
        f"Birth: {self.birth_time}, Rejected: {self.rejected}, " \
        f"Server ID: {self.served_by}, " \
        f"Service Time: {self.service_time}, " \
        f"Death: {self.death_time}"

def __repr__(self) -> str:
    return f"CustomerObject({self.id}," \
        f"{self.birth_time})"

def to_csv(self) -> str:
    """Represent the current state of the customer in CSV format:
    the structure is as follows:
    ID,Priority[not used for this simulation],Birth Time,Death Time,
    Rejected,ServerID(None if rejected), Service Time(None if rejected),
    Death Time
    """
    return f"{self.id},0,{self.birth_time}," \
    f"{self.rejected},{self.served_by},{self.service_time},{self.death_time}"
```

Fig. 4. Customer alternate representations.

The alternate representations of each customer instance allow the information to be clearly described within a string format. Such information is frequently outputted into various files in the directory to allow for users to easily see how the program is running.

2. *UniversalServer*

```
class UniversalServer:
    """A Class representing all servers within the MMCC queue

    @attributes:
        id      -- the unique server id
        idle    -- whether the server is idle
        current_customer    -- The current customer object being served
        serve_time      -- the total time spent serving customers
        idle_time       -- the total time spent idle
        last_update_time    -- the last time since update()
        cust_served     -- the ammount of customers served
        rands           -- the list of random numbers used for service time
    """
    id : int
    idle : bool = True
    current_customer : Customer | None = None

    serve_time : int = 0
    idle_time : int = 0
    last_update_time : int = 0
    cust_served : int = 0

    rands = []
```

Fig. 5. The UniversalServer class attributes.

The UniversalServer class contains all information for each server including the ID, whether the server is idle, the current customer, the serve time, the idle time, since the class was lost updated, and the number of customers served. Additionally, an array of rands, which contain the random times regarding how long each service takes, is included. In this class, it is worth noting that the rands attribute does change during run-time.

```
    def __init__(self, given_id : int) -> None:
        self.id = given_id

    def set_serve_time(self, rands : List[int]):
        """Set the given random array to the servers internal memory.
        The memory is removed throughout the runtime of the program.

        @parameters:
            rands -- The time steps required to complete each service
        """
        self.rands = rands  # Python passes arrays by reference so this is just
                            # for ease of access / code clarity
        logging.info("Set server: {%s random string", repr(self))
```

Fig. 6. UniversalServer initialisation and setting rands.

The initialization of the server simply takes in a given ID and attributes it to each server. However, the set serve time method takes in a list of rands produced by the MMCCimulation class and then attributes it to the server.

This method first updates the values within the server class as, for each server, it is not updated for each simulation tick. Then it gets the time to serve a given customer using the rand's variable. It calls the customer.serve function. It updates its new internal attributes. Then it outputs a log and returns the time the service of the customer will end.

Upon the server finishing its service, the program will log the event and then kill the current customer, set the current customer to none and update its variables. The update function simply checks

```python
def serve(self, customer : Customer, time : int) -> int:
    """Set a server to serve a customer

    @parameters:
        customer -- The customer to be served by the server
        time -- The time of the service start

    @returns:
        The time thaat the server completes its service
    """
    self.update(time)
    time_to_serve = self.rands.pop(0)
    customer.serve(self, time_to_serve)
    self.current_customer = customer
    self.idle = False
    self.cust_served += 1
    string = f"Assigned server: {repr(self)} customer: {repr(customer)}. Time to serve: " \
             f"{time_to_serve}, finish time: {time + time_to_serve}"
    logging.info(string)

    return time_to_serve + time
```

Fig. 7. UniversalServer serve function.

```python
def finish_serve(self, time : int) -> None:
    """Kill the current customer and clean-up interal attributes at
    time of completed service

    @parameters:
        time -- the simulation time of the completed service
    """
    logging.info("Completed server: %s service of customer: %s",
                 repr(self),
                 repr(self.current_customer))
    self.current_customer.kill(time)
    self.current_customer = None
    self.update(time)
    self.idle = True

def update(self, time : int) -> None:
    """Update time dependent attributes when changing idle state

    @parameters:
        time -- the simulation time of the update.
    """
    logging.info("Updating server: %s internal attributes", repr(self))
    if self.idle:
        self.idle_time += (time-self.last_update_time)
        self.last_update_time = time
        return
    self.serve_time += (time-self.last_update_time)
    self.last_update_time = time
```

Fig. 8. UniversalServer finish serving and update functions.

the current state of the server and then adds the time that has passed since its last update to the relevant attribute.

As with the customer alternative representations, the server representations also contain the server ID, but they contain the current state of the server, its total idle time, the current customer representation if it has one, the number of customers served, and the total service time. Additionally, a CSV format is included to allow for easy checking of the simulation after output. The CSV format may also be used for graphing purposes.

```python
def __str__(self) -> str:
    return f"Server ID: {self.id} " \
        f"Idle: {self.idle}, Idle time: {self.idle_time}, " \
        f"Current Customer: {repr(self.current_customer)}, " \
        f"Customers Served: {self.cust_served}, " \
        f"Total service time: {self.serve_time}"

def __repr__(self) -> str:
    return f"UniversalServer({self.id})"

def to_csv(self) -> str:
    """Convert the server information into csv format"""
    return f"{self.id},0,{self.idle},{self.idle_time},{self.cust_served}," \
        f"{self.serve_time}"
```

Fig. 9. UniversalServer alternate representations.

3. MMCCSimulation

```python
class MMCCSimulation:
    """The MMCCSimulation body. A simulation for a service and customers,
    where once all services are filled, customers are rejected completely.
    Used for Modelling and Simulation CA1, assignment 1.

    @attributes:
        customer_count -- The number of customers the simulation runs for
        server_count -- The number of servers for the simulation
        service_avg -- The exponential-average service time
        arrival_rate -- The exponential-average customer arrival time
        rand_arrays -- Arrays containing the random values above.
                    -- Note, as apposed to server-rands, this does not change
                    -- at run time.
        servers -- The list of all active servers
        customers -- The list of all Alive and Killed customers
        customer_birth_times -- The times a customer is born
        next_events -- An array of all future staged event timings
    """
    customer_count : int
    server_count : int
    service_avg : int
    arrival_rate : float

    rand_arrays : List[List[int]]
        # The first index contains the times of birth for the customers,
        # The following determines the time of processing for the servers
    time : int

    servers : List[UniversalServer]
    customers : List[Customer]

    customer_birth_times : List[int]
    next_events : List[int]
```

Fig. 10. MMCCSimulation attributes.

The MMCCSimulation class is where the majority of the simulation logic appears. It contains information as to the amount of customers requested, the amount of service defined, the service average, and the arrival rate. It contains the current time of the simulation the arrays used to determine the randomness of the simulation, a list of the current servers, and a list of the created customers. It also contains a list containing the times that a customer is born, and a list of times regarding when each upcoming event is staged.

```
def __init__(self,
             customer_count : int,
             server_count : int,
             service_avg: int,
             arrival_rate: float,
             start = round(timeMod.time())) -> None:
    logging.info("Initialising simulation")
    self.customer_count = customer_count
    self.server_count = server_count
    self.service_avg = service_avg
    self.arrival_rate = arrival_rate
    self.start_time = start

    self.next_events = []
    self.servers = []
    self.customers = []
    self.time = 0

    self.set_rand_array()
    self.create_servers()

    for array in self.rand_arrays:
        logging.info("Rands: %s", str(array))
```

Fig. 11. MMCCSimulation initialisation..

When the MMCCSimulation class is initiated it sets all attributes to those as given in the initialization, it sets the next event, the servers, and the customers lists to be empty. Additionally sets the time of the simulation to 0. It also creates both the servers and the random arrays.

```
def set_rand_array(self) -> None:
    """Set the random arrays to the interal attributes of the class"""
    self.rand_arrays = []
    logging.info("Setting random arrays for the simulation")
    # Generate Customers

    self.rand_arrays.append(discrete_exponential(1/self.arrival_rate, self.customer_count))

    self.customer_birth_times = self.rand_arrays[0].copy()
    self.next_events.append(self.customer_birth_times.pop(0))
        # Add the first birth of customer to the event list
    self.produce_server_rand_arrays()
    logging.info("Finished random arrays for the simulation")

def produce_server_rand_arrays(self):
    """Create the server random arrays values"""
    for _ in range(self.server_count):
        self.rand_arrays.append(discrete_exponential(self.service_avg, self.customer_count))

    logging.info("Finished setting server random arrays")
```

Fig. 12. MMCCSimulation methods called to produce the random numbers.

The methods used to produce random functions are the set_rand_array and produce_server_rand_arrays functions. The set_rand_array function first sets the attribute to an empty list then it appends a discrete exponential which is created using the arrival rate of customers. Then it sets the customer_birth_times attribute to be a copy of the list created. The reason we copy the list is, as Python treats list passing by reference, any changes to a list which is within another, will change the original. Therefore as we want to maintain the rand_arrays for future reference, we do not wish to change the value during runtime. The next_event attribute is then appended with the first customer birth time, and the simulation then creates the server random arrays. Producing the server random arrays is fairly simple as we simply create a discrete exponential array of random numbers which are centred around the service average for each of the servers.

```python
def create_servers(self) -> None:
    """Initialise all the servers for the class"""
    self.servers = [UniversalServer(x) for x in range(self.server_count)]
    for i, serv in enumerate(self.servers):
        serv.rands = self.rand_arrays[i+1].copy()
        self.next_events.append(999999999)
        logging.info("Finished initialising server: %s", repr(serv))

def birth_customer(self, priority : int | None = None) -> None:
    """Initialise a customer at a given time. Only produced one instance
    as it is only run when a customer 'joins' the simulation Priority included as interface"""
    customer = Customer(self.time, len(self.customers))
    self.customers.append(customer)

    logging.info("Created customer: %s", repr(customer))
    # As we don't have a queue, if every server is full, the customer is turned away
    self.assign_customer(customer)
    return customer
```

Fig. 13. MMCCSimulation methods creating servers and customers.

To create all servers in the simulation we simply create a universal server for a given unique ID. Then we attribute each server's random array to one created in the simulation. Finally, we set the next event corresponding to the server's event to be a significant time in the future. When we create a customer, we give the initialization the time the server was created and the length of the list of the currently created customers which corresponds to an identifier. Then as the simulation does not have a queue, we immediately attempt to assign the customer to a server.

```python
def assign_customer(self, customer : Customer) -> bool:
    """Assign a customer to a server. The current method choses the server
    with the lowest ID. If no available server is found, kills the customer

    @parameters:
        customer -- The customer to be served
    """
    available_servers = self.get_available_servers(customer)

    if len(available_servers) == 0:
        logging.info("No available servers at this time for customer %s", repr(customer))
        customer.reject(self.time)
        return False

    # The manner of choosing which server to pick can be assigned here, but
    # for now, we'll just go for the lowest ID
    logging.info("Available servers found for customer %s", repr(customer))
    chosen_serv = available_servers[0]
    next_event_time = chosen_serv.serve(customer, self.time)
    self.next_events[self.servers.index(chosen_serv)+1] = next_event_time
    return True
```

Fig. 14. MMCCSimulation assigning customers.

When assigning customers the program first gets all available servers which is simply done by checking which servers are currently idle. Then if the length of the available services is zero, the customer is rejected. Otherwise, the customer is assigned the first available server, the program finds the next available time using the UniversalServer.serve function, and then changes the next event representing the server to be the time the server is finished.

As the simulation does not render each tick individually, the program simply jumps to the next stage event. This is done by getting the information as to the next event which is the minimum of all the values stored within the next events attribute. Then it gets the list of all indices which have the event time as equal to the time being jumped to and returns the list of these indices.

The MMCCSimulation run function first checks to make sure the staged events are in a reasonable time, then gets the list of indices which are staged currently. It then checks whether the indices correspond to the birth of a customer. If so it creates a customer object and checks whether it has

```python
def jump_next_event(self) -> List[int]:
    """Move the simulation time to the next staged event. Probably less
    efficient than a for loop, but hey ho.

    @returns:
        A list of all the indices of staged events that has been jumped to
    """
    next_time = min(self.next_events)
    logging.info("Moving time forward: %s -> %s", self.time, next_time)
    self.time = next_time

    staged_events : List[int] = []
    for i, event_time in enumerate(self.next_events):
        if event_time == self.time:
            staged_events.append(i)

    logging.info("Staged event(s) found at index %s", str(staged_events))
    return staged_events
```

Fig. 15. MMCCSimulation time jump method.

```python
def run(self):
    """Run the MMCC Simulation with the given parameters
    """
    logging.info("Running simulation")
    # While there are still staged events...
    while min(self.next_events) < 999999999:
        staged_events = self.jump_next_event()

        if staged_events[0] == 0:
            self.birth_customer()

            if self.customer_birth_times:
                self.next_events[0] = self.time + self.customer_birth_times.pop(0)
            else:
                self.next_events[0] = 999999999

        for index in staged_events:
            if index == 0:
                continue

            self.servers[index-1].finish_serve(self.time)
            self.next_events[index] = 999999999

    for server in self.servers:
        server.update(self.time)
        # Update to include end of service stint.

    logging.info("Simulation Completed:")
    logging.info("Final states:")
    logging.info("Simulation time: %s", self.time)
    logging.info("Customers:")
    for cust in self.customers:
        logging.info(str(cust))

    logging.info("Servers:")
    for serv in self.servers:
        logging.info(str(serv))

    logging.info("Loss rate: %s", str(self.find_loss_rate()))
    return
```

Fig. 16. MMCCSimulation run logic.

exceeded the number of customers to be spawned. Then it checks for the indices representing a server event. If so it makes the server corresponding to the event finish the serving and sets the next event time to be significantly far in the future.

1.2. M1-M2/M/C/C Queue Simulation

The M1-M2/M/C/C queue simulation relies heavily on inheritance from the MMCC simulation. All but one of the classes defined in this section contain inheritance from the MMCC simulation. The only object that does not inherit from the previous file is the PriorityMismatchError.

```
class PriorityMismatchError(Exception):
    """Exception raised when a customer is being served by a server
    which they should not have access to.

    @attributes:
        customer_priority -- the customers priority
        server_priority -- the servers priority
        message -- explaination of the error
    """

    def __init__(self, customer_priority : int, server_priority : int) -> None:
        self.customer_priority = customer_priority
        self.server_priority = server_priority

        self.message =  f"""Customer does not have adequate priority to be served
            by this server. ({customer_priority} < {server_priority})
            """.strip("\n", "")
        super().__init__(self.message)
```

Fig. 17. PriorityMisMatch Error

The priority mismatch error is not invoked during the standard runtime of my implementation of the project. Rather, it is created to provide the user with an error if they are making adaptations to the software and in doing so they create an error where the server is serving a customer when it does have the correct priority.

```
class PriorityCustomer(Customer):
    """Overriding the Customer class adding priority functionality and service
    A customer can only use servers of a less than or equal to priority

    @parameters:
        priority -- A represention of which rank of servers a customer can use
    """
    priority : int
    def __init__(self, time: int, given_id: int, priority: int = 0) -> None:
        self.priority = priority
        super().__init__(time, given_id)

    def serve(self, server : 'PriorityServer', time : int):
        """Overrides the Customer serve method. Compares the priority of
        the customer and the server, and judges if they are compatable

        @parameters:
            server -- The priority server to serve the customer
            time -- The simulation time of the start of the service

        @raises:
            PriorityMismatchError -- Where server and customer priority is incompatable
        """
        if server.priority > self.priority:
            raise PriorityMismatchError(self.priority, server.priority)
        super().serve(server, time)
```

Fig. 18. PriorityCustomer Class

The priority customer class relies heavily on the inheritance from the previous Customer class. The only change to it is the inclusion of a new priority attribute that is set during the initialization of the priority customer object. There is a minor change to the serve method as it checks for the server's

priority and compares it to the customer's priority to guarantee that the customer has the right value. If it is not correct then the priority mismatch error is raised.

```
class PriorityServer(UniversalServer):
    """Overrides the UniversalServer class adding priority functionality
    a server can serve only serve customers with a >= priority

    @parameters:
        priority -- A representation of which customers the server should serve"""
    priority : int

    def __init__(self, given_id: int, priority: int = 0) -> None:
        self.priority = priority
        super().__init__(given_id)
```

Fig. 19. Enter Caption

There is a very minor change to the priority server as it allows for priority functionality. The priority attribute is added to the object and written to when initializing the instance. This change is then outputted during the standard alternative representations.

There is a few changes to the attributes of the M1M2MCCSimulation class as opposed to the previous MMCCSimulation. The server_amount now carries an array where the index corresponds to the priority of each server and the value at that index response corresponds to the number of servers at that priority. Service average allows different servers to have different rates, although, for this implementation, all rates will be constant and global. ”servers” becomes a list of PriorityServers and ”customers” becomes a list of PriorityCustomers. During initialisation, these new values are set and the standard __init__ function is then run. Additionally, towards the end, the initialization appends the first birth of higher-priority customers to the next event list.

```
class M1M2MCCSimulation(MMCCSimulation):
    """ The object containing the logic for a M1/M2/M/C/C Queue

    @attributes:
        server_ammounts     -- A list where the index is the priority and the value is
                               the ammount of servers for that priority
        service_avg         -- A list of the average service time for each server priority
        servers             -- Altered to [PriorityServer] type
        customers           -- Altered to [PriorityCustomer] type
    """
    server_ammounts : List[int] # [14,2]
    service_avg : List[int] # Override old type as prioirities could have diff rates
    servers : List[PriorityServer] = []
    customers : List[PriorityCustomer]

    def __init__(self,
                 customer_count : int,
                 server_ammounts : List[int],
                 service_avg : List[float],
                 arrival_rates : List[float],
                 start_time : int = round(time_module.time())
                 ) -> None:
        self.server_ammounts = server_ammounts
        self.arrival_rates = arrival_rates
        super().__init__(customer_count,
                         sum(server_ammounts),
                         service_avg,
                         arrival_rates,
                         start_time)

        logging.info("server ammounts: %s", str(server_ammounts))

        for array in self.customer_birth_times[1:]:
            self.next_events.append(array.pop(0))
```

Fig. 20. M1M2MCCSimulation initialisation and attributes

```python
def set_rand_array(self) -> None:
    """Set the random arrays for the customer creation and the servers"""
    self.rand_arrays = []

    if self.arrival_rates[0] == 0.0:
        self.rand_arrays.append([99999999999 for _ in range(self.customer_count)])
    else:
        self.rand_arrays.append(discrete_exponential(1/self.arrival_rates[0], self.customer_count))
    self.customer_birth_times : List[List[int]] = []
    self.customer_birth_times.append(self.rand_arrays[0].copy())

    self.next_events.append(self.customer_birth_times[0].pop(0))
    self.produce_server_rand_arrays()

    for rate in self.arrival_rates[1:]:
        if rate == 0:
            self.rand_arrays.append([99999999999 for _ in range(self.customer_count)])
        else:
            self.rand_arrays.append(discrete_exponential(1/rate, self.customer_count))
        self.customer_birth_times.append(self.rand_arrays[-1].copy())
```

Fig. 21. M1M2MCCSimulation set random array (called by __init__)

```python
def produce_server_rand_arrays(self):
    for priority, ammount in enumerate(self.server_ammounts):
        for _ in range(ammount):
            self.rand_arrays.append(discrete_exponential(
                self.service_avg[priority],
                self.customer_count))
```

Fig. 22. M1M2MCCSimulation creating servers rands

The set random array method is run when calling the initialization. First, it produces the random arrays for the base priority customers, setting the time to create a significantly high value if there are no customers of that type. Then it produces the lists of random numbers for all the servers. Then it produces the times of the customer creations for any priority customers, again setting the time significantly far away if there are no customers of that priority.

The method of creating the servers' randoms is very simple as it goes through the server amount given and for each index creates the number of servers required as stated in the array.

To create the servers it simply goes through the number of servers given and creates the specific number of priority servers as stated in the array. Once completed, it assigns the random arrays to each server, and then sets the corresponding event times of the simulation to be significantly far away.

```python
    def create_servers(self):
        curr_id = 0
        for priority, ammount in enumerate(self.server_ammounts):
            for x in range(ammount):
                self.servers.append(PriorityServer(curr_id + x, priority))
            curr_id += 1
        for i, server in enumerate(self.servers):
            server.rands = self.rand_arrays[i+1].copy()
            self.next_events.append(999999999)

    def birth_customer(self, priority : int) -> None:
        customer = PriorityCustomer(self.time, len(self.customers), priority)
        self.customers.append(customer)

        # As we don't have a queue, if every server is full, the customer is turned away
        self.assign_customer(customer)
        return customer
```

Fig. 23. M1M2MCCSimulation creating servers and customers

```
def get_available_servers(self, customer : PriorityCustomer = None) -> List[UniversalServer]:
    if customer is None:
        raise TypeError("customer can not be None")
    output = [server for server in self.servers if (
        server.idle and server.priority <= customer.priority)]
    return output
```

Fig. 24. M1M2MCCSimulation finding available servers

```
def run(self):
    while min(self.next_events) < 999999:
        staged_events = self.jump_next_event()

        for index in staged_events:
            if 1 <= index < 1 + len(self.servers):
                self.servers[index-1].finish_serve(self.time)
                self.next_events[index] = 99999999
                continue

            if index == 0:
                priority = index
            else:
                priority = index - len(self.servers)


            self.birth_customer(priority)

            if self.customer_birth_times[priority] and len(self.customers)<self.customer_count-1:
                self.next_events[index] = self.time + self.customer_birth_times[priority].pop(0)
            else:
                self.next_events[index] = 999999999
```

Fig. 25. M1M2MCCSimulation run logic

For creating customers, it simply creates a customer with a given time, priority, and ID, then attempts to assign them to a server.

To find available servers, it simply checks all servers in the server list in simulation as to whether they are idle and have a priority less than or equal to the current customer.

The run logic for the M1M2MCCSimulation simply checks if there are events staged, and then goes through those events. If the index corresponds to a server, the server finishes and sets the corresponding next event time to be significantly far away. If it is found to correspond to a customer, the program calculates the priority using the index value and births a customer. The program then checks to make sure there are adequate birth customer times values and whether the number of customers created exceeds the limit for the simulation.

2. Testing, Validation and Diagrams

This section includes the relevant details as to the testing and validation of the models provided. Additionally, it includes a brief discussion on the diagrams produced while running the algorithm. Be validation consists of a comparison between the output log and a hand-computed simulation for both M/M/C/C simulations and M1-M2/M/C/c simulations. Additionally, for M/M/C/C Queue simulations diagrams showing the performance results of the program are included.

2.1. M/M/C/C Queue simulations

1. Hand-computation evaluation

The following random values for generating customers and service times were produced when running the program. The number of servers is 4, the number of customers is 15, the average service time is 100 seconds, and the average arrival time is 10 seconds.

Customer Intervals	3	2	23	4	3	4	1	2	7	27	7	9	1	31	10
Server 0	45	73	33	24	319	20	276	120	25	322	53	31	48	171	144
Server 1	33	171	76	14	142	97	31	45	46	49	26	84	3	14	146
Server 2	74	70	102	64	155	20	25	197	71	57	479	100	143	93	253
Server 3	56	368	92	58	167	105	119	128	70	101	70	104	192	96	134

The following is a hand-produced simulation of the events of an M/M/C/C Queue with the previous values representing the customers.

Customer ID:	Birth Time	Rejected	Server ID	Service Time	Death
0	3	FALSE	0	45	48
1	5	FALSE	1	33	38
2	28	FALSE	2	74	102
3	32	FALSE	3	56	88
4	35	TRUE			35
5	39	FALSE	1	171	210
6	40	TRUE			40
7	42	TRUE			42
8	49	FALSE	0	73	122
9	76	TRUE			76
10	83	TRUE			83
11	92	FALSE	3	368	460
12	93	TRUE			93
13	124	FALSE	0	33	157
14	134	FALSE	2	70	204

26 is a screenshot of the results/no-rank/customer/validation.csv file produced by the algorithm.

As is clear, the results of the algorithm are identical to a hand-computed simulation of the M/M/C/C queue, implementing the blocking and freeing of servers.

Additionally, to guarantee that server information is stored correctly, as a result of the hand-calculated simulation, the values for the created servers are as follows.

Server ID	**Service Time**	**Customers Served**
0	151	3
1	204	2
2	144	2
3	424	2

27 is the resultant CSV file from completing the algorithm simulation. Again, as is clear, the values perfectly match the results created by the hand-processed simulation. Therefore, we can consider

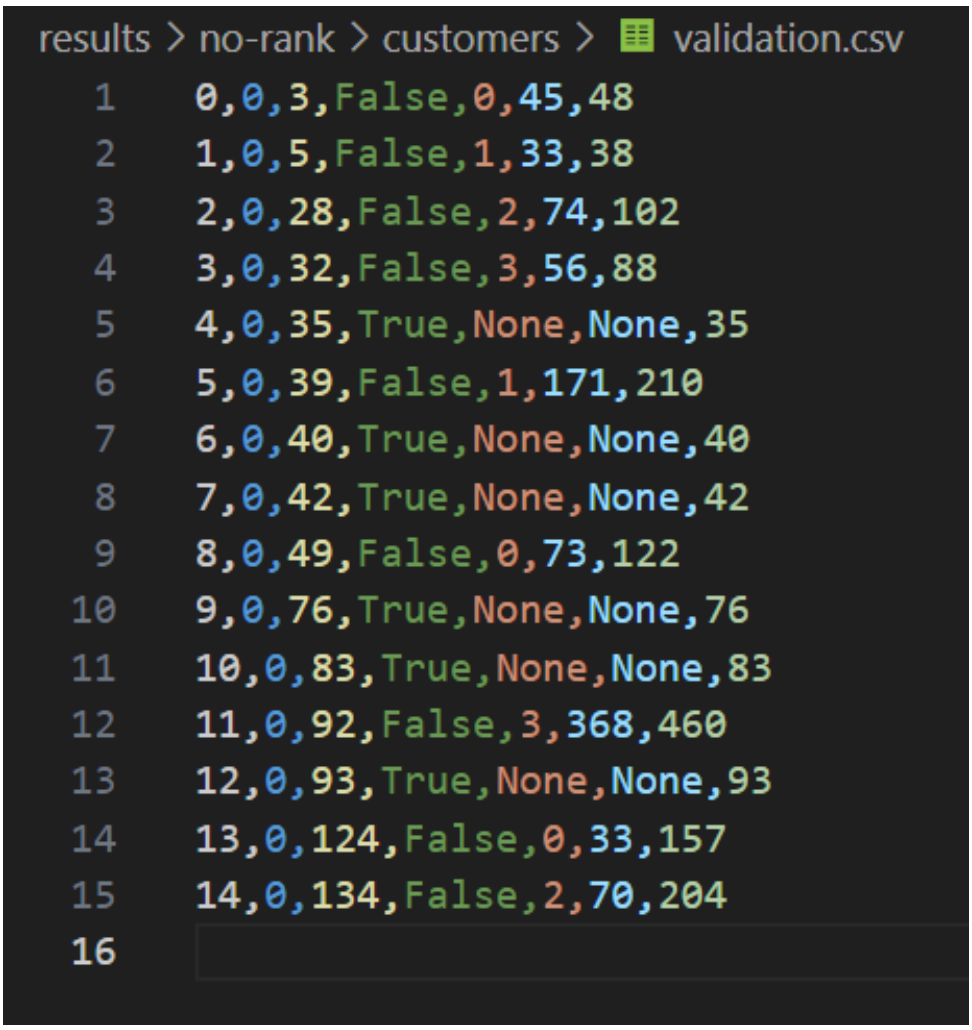

Fig. 26. results/no-rank/customer/validation.csv: Customer simulation results

that the algorithmic simulation for the M/M/C/C queue has adequate functionality to appropriately work.

Fig. 27. results/no-rank/server/validation.csv: Server simulation results. Column 7 (cyan) is the total service time

$$P_c = \frac{(\frac{\lambda}{\mu})^c / c!}{\sum_{k=0}^{c} (\frac{\lambda}{\mu})^k / k!}$$

Fig. 28. The equation to model the loss rate (P_c) for an M/M/C/C queue, where λ is the arrival rate, μ is the service rate, and c is the maximum capacity.

```python
def loss_rate(ar : float, sr: float, max_servers : int) -> float:
    numerator = ((ar/sr) ** max_servers) / math.factorial(max_servers)
    denominator = 0
    for k in range(max_servers + 1):
        temp = (ar/sr) ** k
        temp = temp / math.factorial(k)
        denominator += temp

    return numerator / denominator
```

Fig. 29. The algorithmic implementation of 28

2. *Blocking rate analysis:*

To detect inconsistency with the random number generation of the simulation, and to guarantee that all sections of the M/M/C/C queue are working as intended, we can use the probabilities created by representing the queue as a steady state diagram in order to get an analytical result for the anticipated blocking rate. The formula to calculate the anticipated blocking rate for a given arrival rate, service rate, and maximum capacity is shown in 28.

The program first calculates the analytical blocking rate using the average service length and the arrival rate for a range of values between 0.01 and 0.1. The algorithm within the program to compute these values is shown in 29.

To produce the values which come from the simulation, as the program heavily relies on random number generation, for each value to be tested, a simulation with 2,000 customers is created and run 25 times and the average blocking rate is calculated. Over a span of 50,000 customers, the impact of random number generation should be drastically decreased and therefore allow appropriate results and graphs to be formed. The output of the graph, showing both the analytical and empirical simulation blocking rates, is found in 30

The graph clearly shows that the empirical simulation follows the analytical results for the blocking probability. However, it is worth noting that the standard simulation seems to have a decreased blocking rate over the analytical block probability for values over 0.06. A potential explanation for this is the discretization of the exponential distribution. As the simulation does not allow multiple customers to arrive at the same time, there is a potential that small values of customer intervals are being incorrectly rounded as a result of the large time scale. A method to test this is to increase the time scale of both the arrival rate and service time. Increasing the timestep by 10x produces the graph 31. This graph more accurately follows the analytical solution and therefore it is likely that the decrease in blocking probability is a result of the discretization of the exponential distribution function.

3. *Server utilisation heat-map*

Another method of displaying simulation data is using a heat map representing the proportion that a server is in use throughout this simulation. The method for displaying this data has been included in the program. However, to prevent artefacts generated from the random data created through the simulation, the program completes 25 runs before producing the graph. The program then takes the

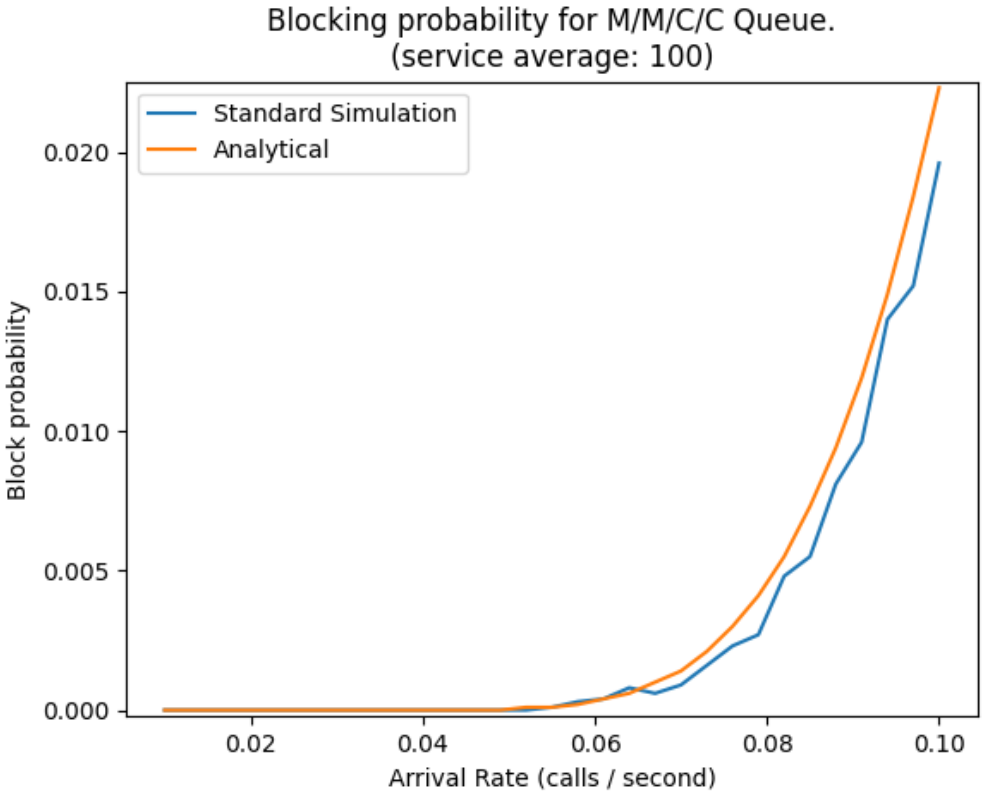

Fig. 30. The results of an imperial simulation of an M/M/C/C queue with 16 servers.

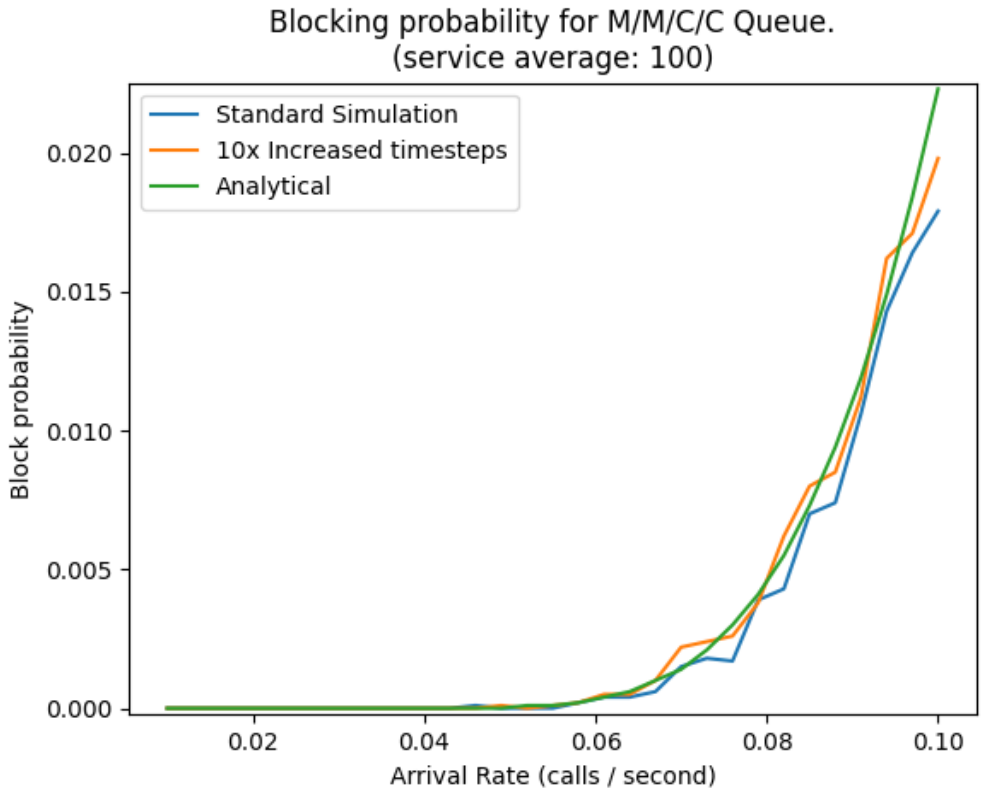

Fig. 31. The results of an imperial simulation of an M/M/C/C queue with 16 servers. Including a simulation where the timestep increased by 10x

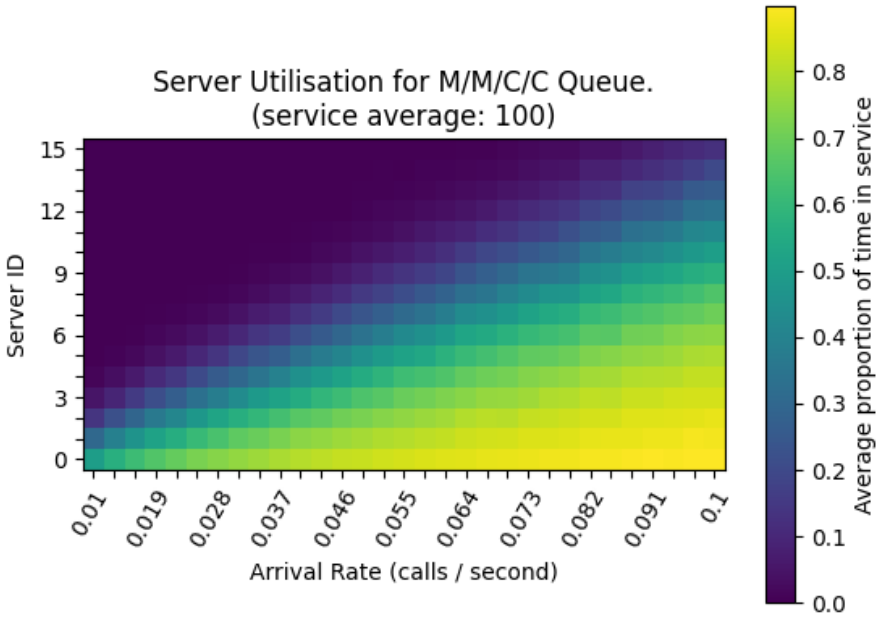

Fig. 32. Server utilisation heat map of an M/M/C/C queue with 16 servers

average server utility for each server within the algorithm and produces a heat map. There is result of the heat map can be seen in 32. As it is clear, low arrival rates cause a decrease in server utilization. Additionally, for those low values, very few servers are even active throughout the whole program. For arrival rates of not 0.01, the servers with an ID greater than 6 appear to have never been used. This is in comparison to the 0.1 arrival rate, where all servers have seen consistent use and servers with an ID less than 5 are constantly being used between 75% to 90% of the time.

2.2. M1-M2/M/C/C queue validation

In a similar method to the M/M/C/C queue validation, the queue implementation can be validated using the random number generated by the algorithm to hand-compute a simulation, and then compare the results against those created by the algorithm.

The random numbers generated by the algorithm for a base customer interval, 3 base priority servers, 2 high priority servers, and a high priority customer interval are shown in 2 2.2.

B Customer	3	1	12	11	40	12	4	13	2	3
B Server 0	21	87	13	35	39	42	26	44	14	21
B Server 1	165	68	3	29	10	21	5	21	48	5
B Server 2	85	33	17	54	46	27	19	1	28	9
P Server 0	53	11	26	116	23	94	106	24	26	225
P Server 1	16	29	10	279	18	24	60	38	55	1
P Customer	29	15	4	14	16	18	4	3	5	7

The following is a result of a 10-customer simulation of the M1M2/M/C/C problem described above.

is the CSV format of the customers' results from the program implementation. The CSV format contains identical information to the results produced from the hand-computed simulation. Therefore, I can confidently say that the algorithm correctly implements both queuing systems.

3. Problem implementation

All problems and functionality can be accessed using the main.py program within the /src/ directory. This file will produce a very basic command line interface. From here, you can access the individual

Customer ID	Priority	Birth time	Rejected	Served by	Service time	Death time
0	0	3	FALSE	0	21	24
1	0	4	FALSE	1	165	169
2	0	16	FALSE	2	85	101
3	0	27	FALSE	0	87	114
4	1	29	FALSE	3	53	82
5	1	44	FALSE	4	16	60
6	1	48	TRUE			48
7	1	62	FALSE	4	29	91
8	0	67	TRUE			67
9	1	78	TRUE			78

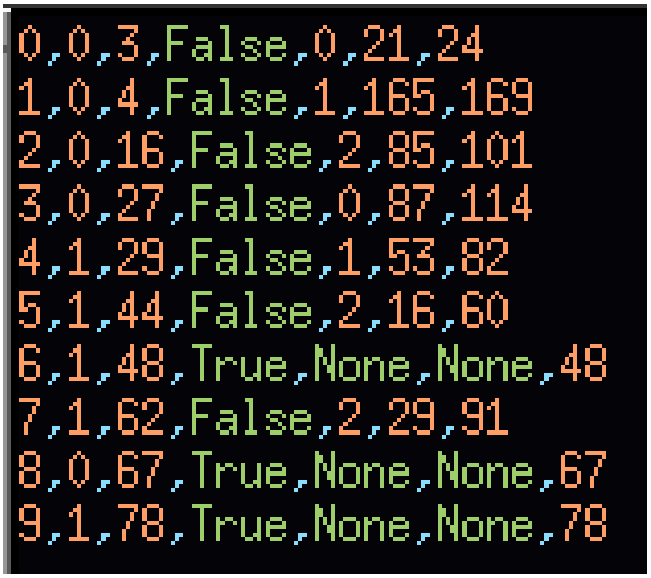

Fig. 33. M1-M2/M/C/C Customer validation CSV file. This time in NeoVim. Note the server IDs are not entirely unique, as servers of different priorities can have the same ID. However, accounting for that information, all servers are appropriately accessed and their information is correctly stored.

functions which produce answers to the questions. Additionally, the user can generate graphs based on the simulation or analytical performance. These graphs have already been included in this report.

3.1. Exercise 1.3

Exercise 1.3 has been implemented in 2 different ways. The first is a simulation-based approach, and the second utilises the steady state loss rate equation 28. Within the menu, option 4 utilises the simulation to attempt to find a value where the block rate is significantly close to 0.01, and uses

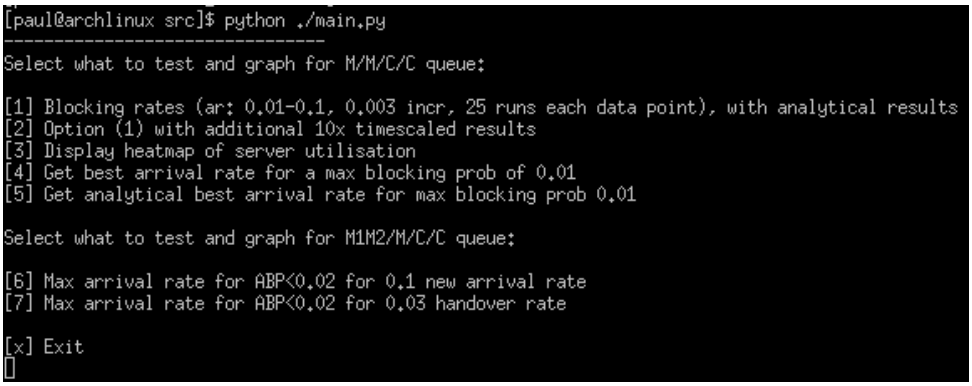

Fig. 34. main.py CLI menu

```python
def find_max_value(max_blocking_prob : float = 0.01) -> None:

    values : List[float] = []
    for _ in range(10):
        values.append(mmcc_max_value(max_blocking_prob))

    min_val = min(values)
    max_val = max(values)

    print(f"Simulation puts the arrival rate between {min_val} and {max_val}")

def mmcc_max_value(max_blocking_prob : float = 0.01) -> float:
    """Implement a binary search of the simulation to find the highest arrival
    rate where the blocking probability does not exceed the max_blocking_prob

    @parameters:
        max_blocking_prob -- The target blocking probability to get
    """
    test_arrival_rate : float = 0.5
    cycle : int = 2
    epsilon : float = 0.0001

    while True:

        value : float = get_average_loss_rate(test_arrival_rate)

        if (value + epsilon) < max_blocking_prob:
            test_arrival_rate += 0.5**cycle
        elif (value - epsilon) > max_blocking_prob:
            test_arrival_rate -= 0.5**cycle
        elif cycle > 30:
            print("Could not resolve answer, restarting...")
            test_arrival_rate = 0.5
            cycle = 1
            epsilon *= 2
        else:
            print(f"Found block rate of {round(value, 4)}, for arrival rate {test_arrival_rate}")
            return test_arrival_rate

        cycle += 1
```

Fig. 35. Functions used to find the maximum arrival rate for a given loss rate for a M/M/C/C

```python
def find_for_loss_rate(sr : float, max_servers : int, search_br : float) -> float:

    ar = 0.5
    cycle = 2
    epsilon = 0.00001
    while True:
        value = loss_rate(ar, sr, max_servers)
        if value + epsilon < search_br:
            ar += 0.5 ** cycle
        elif value - epsilon > search_br:
            ar -= 0.5 ** cycle
        else:
            return ar

        cycle += 1
```

Fig. 36. Functions used to find the maximum arrival rate for a given loss rate for a M/M/C/C using the analytical method

multiple runs to reduce the impact of random numbers for the final value.

The program utilises a binary search over all possible arrival rates from 0.0 to 0.99. If the resulting average loss rate if greater than the target, the test arrival rate is decreased, if not, the value is increased. Once the value get close enough to the target value (we cannot test for an exact match due to floating point arithmetic), we store the value in a list. Once this has been completed 10 times, the program reports the value to the user as a range of values it feels confident are representative of the maximum possible arrival rate for a given target blocking probability.

The analytical method implements a similar binary search, however, this time the program does not have to repeat runs as the analytical formula is deterministic. Therefore, the implementation is as follows.

```python
def question2_2(arrival_rates : List[float | None], target_abp : float) -> float:
    none_indices : List[int] = []

    for i, rate in enumerate(arrival_rates):
        if rate is None:
            none_indices.append(i)

    if len(none_indices) != 1:
        raise ValueError("arrival_rates must contain only one none value")

    arrival_rates[none_indices[0]] = 0.0
    if get_average_abp(arrival_rates) > target_abp:
        print("Target ABP is not possible to achieve as set arrival rates")
        print("already cause the ABP to be higher than the target")
        return

    cycle : int = 2
    epsilon : float = 0.0001
    arrival_rates[none_indices[0]] = 0.5

    while True:
        abp : float = get_average_abp(arrival_rates)

        if abp + epsilon < target_abp:
            arrival_rates[none_indices[0]] += 0.5**cycle
        elif abp - epsilon > target_abp:
            arrival_rates[none_indices[0]] -= 0.5**cycle
        else:
            print(f"Suitable arrival rates found: {str(arrival_rates)}")
            return arrival_rates

        if cycle > 30:
            epsilon *= 2
            arrival_rates[none_indices[0]] = 0.5
            cycle = 1

        cycle += 1
```

Fig. 37. The function which determines the maximum of a selected arrival rate where the hole simulation does not exceed a given ABP.

3.2. Exercise 2.2 / 2.3

These exercises are grouped together as the implementation allows for both questions to be completed by the same algorithm, as shown in 37. This method first finds the appropriate arrival rate to change. Then, the function does a check to make sure that, even if the selected arrival rate is 0, the simulation does not exceed the given ABP. This is important as it prevents the program from continually attempting to search for an impossible answer. Then the program implements a similar method to Exercise 1.3, where a binary search is implemented for the unknown arrival rate, an the value is successively homed in on until the correct answer is found. If the value cannot be found after 30 cycles, the epsilon value is increased to allow a wider range of potential answers. When the answer is found, the program outputs the value to the command line.

The get_average_abp function simply serially creates 50 M1-M2/M/C/C queue simulations with 2000 customers each, and calculates the average aggregate blocking probability over all simulations. The code for ABP average calculation is included in 39. NOTE! The video presentation includes a bug where the ABP calculation is not correctly divided when returning. This is corrected in the image provided.

4. Results

4.1. Exercise 1.3

The results indicate that the values between 0.089 and 0.091 fall within 0.0001 of the blocking probability. Therefore, for a blocking probability of 0.01, I would suggest an arrival rate 0.089 if you must not exceed the 0.01 metric for a considerable period of time. Otherwise, I would suggest a round 0.09, if you only wish to have more days where the value is under 0.01 than over. However, it is worth considering that, as shown before the discretisation of the exponential distribution decreases the average blocking rate, if you are in a more continuous environment, i would suggest staying before 0.0887, as given by the analytical solution, in 40.

```
def get_average_abp(ars : List[float]) -> float:

    abp : float = 0.0
    for _ in range(50):
        sim = M1M2MCCSimulation(
                    2000,
                    SERVERS,
                    SERVICE_AVG,
                    ars
                    )
        sim.run()
        abp += sim.get_abp()

    return abp / 50
```

Fig. 38. The function that calculates an average ABP over a series of M1-M2/M/C/C Simulations.

```
def get_average_abp(ars : List[float]) -> float:

    abp : float = 0.0
    for _ in range(50):
        sim = M1M2MCCSimulation(
                    2000,
                    SERVERS,
                    SERVICE_AVG,
                    ars
                    )
        sim.run()
        abp += sim.get_abp()

    return abp / 50
```

Fig. 39. The command line output for the simulation interpretation of Exercise 1.3

4.2. Exercise 2.2

Utilising the function described earlier, the program outputs the following result shown in 41. This occurs as the arrival rate from the standard new calls already has increased the blocking rate past 0.02. As the formula for the ABP is 10*handover failure + new call failure, unless the handover failure was negative, the ABP would exceed the target value. Therefore, as the algorithm states, a solution for this question can never be acquired, as the target is impossible to reach.

4.3. Exercise 2.3

The output of the main.py program for Exercise 2.3 is shown in 42. The program outputs that for a standard call arrival rate, the program recommends approximately 0.0470. While the program completed its assessment, it is worth noting that, unlike Exercise 1.3, the program did not complete multiple searches. Rather it found the average of 50 simulations, and so is potentially impacted by the

Fig. 40. The command line output for the analytical interpretation of Exercise 1.3

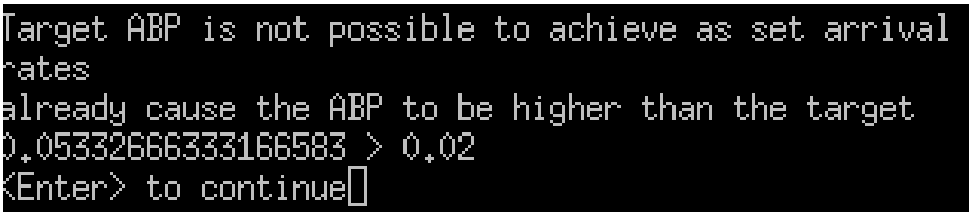

Fig. 41. The command line output for the exercise 2.3

randomness generated when assigning timings for the simulation

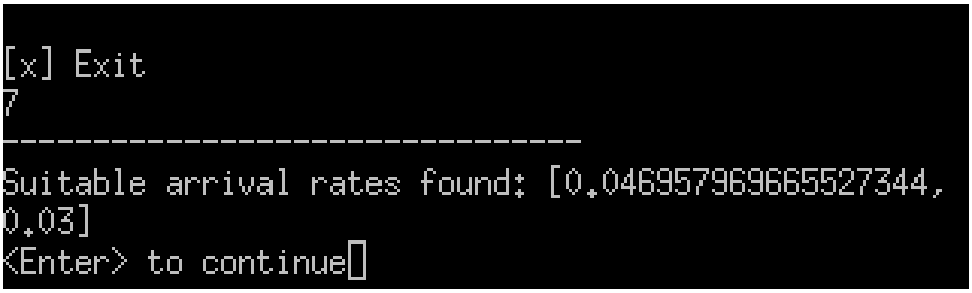

Fig. 42. The command line output for exercise 2.3