Team
Rishi Bhatt
Michelle Cheng
Arterio Rodrigues
Nisagra Kadam

Problem Set 3 Solution

Problem 1a)
The full code for this can be found HERE

We simulated a shared vehicle station where

- Vehicles arrived according to a poisson process with rate $\lambda = 6$
- There are three types of clients requesting bikes:
  - Class 1: members rate $\mu_1 = 3$, fee $K_1 = 0.5$, penalty $c_1 = 1.0$
  - Class 2: members rate $\mu_2 = 1$, fee $K_2 = 0.1$, penalty $c_2 = 0.25$
  - Class 1: members rate $\mu_3 = 4$, fee $K_3 = 1.25$, no penalty

The simulation runs for T = 120 time units start with X(0) = 10 initail vehicles.
We test the simulation with difference number of occurence with the results being

Occurences	Result
10	$565.1
100	$553.793
1000	$552.23
10000	$551.69
100000	$551.71

```
double arrivalTime = randomExponentialNumberGenerator(1.0 / lambda);
double client1ArrivalTime = randomExponentialNumberGenerator(1.0 / client1Rate);
double client2ArrivalTime = randomExponentialNumberGenerator(1.0 / client2Rate);
double client3ArrivalTime = randomExponentialNumberGenerator(1.0 / client3Rate);
```

We use discrete event simulation tracking four event types: bike arrivals and three client requests. Interarrival times are generated using **randomExponentialNumberGenerator** with appropriate rates. At each step, we find the next event (minimum time among all events), advance the clock, update the state (occupancy, penalties, rides), and generate new event times. This continues until time ≥ T.

After time T, the membership revenue is calcualted with
$\text{Membership revenue} = (K_1\mu_1 + K_2\mu_2) * T$
$\text{Ride revenue} = (\text{client3Rides}) * K_3$
$\text{Penalties} = c_1(\text{client1Penalties}) + c_2(\text{client2Penalties})$

We use 10,000 replications as the estimate stabilizes, yielding an estimated net profit of $551.69 since the accuracy of the model didn't show much improvment after this point.

Problem 1b

i)

The full code can be found HERE

By the **Superposition Theorem**- If we merge all the poisson processes we get a superposition of independent Poisson processes with rates $\lambda_{\text{Total}} = \lambda + \mu_1 + \mu_2 + \mu_3 + \lambda = 6 + 3 + 1 + 4 = 14$

The total number of events in the distribution M would be

$M \sim \text{Poisson}(\lambda_{\text{Total}} * T) = \text{Poisson}(14 * 120) = \text{Poisson}(1680)$

Due to **Order Statistics**, given M events in $(0,\ \text{T}]$, the event times are distributed as the order statistics of M uniform random variables on $(0,\text{T}]$.

ii)

To generate M we can use the function in problem-set-2 to generate a random variable using the

$M \sim \text{Poisson}(\lambda_{\text{Total}} * T) = \text{Poisson}(1680)$

```
double generateM(double lambda, double timeInterval) {
  double randomNumber = randomFloatGenerator(0, 1);

  return transformationMethodPoisson(lambda * timeInterval, randomNumber);
}
```

iii)

By **Decomposition Theorem** - Given that an event occurred in the merged process the probability it's of each type is:

- P(Arrival | event) = $\frac{\lambda}{\lambda_{\text{Total}}} = \frac{6}{14}$
- P(Class 1| event) = $\frac{\mu_1}{\lambda_{\text{Total}}} = \frac{3}{14}$
- P(Class 2| event) = $\frac{\mu_2}{\lambda_{\text{Total}}} = \frac{1}{14}$
- P(Class 3| event) = $\frac{\mu_3}{\lambda_{\text{Total}}} = \frac{4}{14}$

Then we can change our code to choose a random event based on these probabilties,

```
  double lambdaTotal = lambda + client1Rate + client2Rate + client3Rate;

  std::vector<double> eventWeights = {lambda, client1Rate, client2Rate,
client3Rate};
  std::vector<double> results = {};
  results.reserve(numberOfReplication);

  for (int i = 0; i < numberOfReplication; i++) {
    int M = generateM(lambdaTotal, timeInterval);

    double time = 0.0;
    int occupancy = initalOccupancy;

    int client3Rides = 0;
    int client1Penalties = 0;
    int client2Penalties = 0;

    for (double j = 0; j < M; j++) {
      int M = generateM(lambdaTotal, timeInterval);
```

Again we choose to use 10,000 replications since the estimation did improve much after this point.

Problem 1c)

Both simulation methods produced nearly identical results being $551.69 with 10,000 replications.

The Discrete Event Simulation

Tracked each event chronologically with exact timestamps. It stores these events and removed event that have happens and add new timestamps to the state. This process is more natural as it mirrors the actually process.

The Retrospective Simulation

This was simpler to implement with a constant M events. Using superposition theorem we where able to merge the Poisson processes and using decomposition theorem randomly assign event types based on the proportional rates.

For time comparision the Retrospective Simulation ran slower while the Discrete Simulation was faster.

Discrete Simulation Time: 1886ms

Retrospective Simulation Time: 11292ms

Problem 2a)

The variables include queue length and server's status (busy/free). The residual clocks keep track of the next arrivals and the departures.

Problem 2b)

To estimate θ by simulating the queue length using discrete event simulation We track the queue length over time and compute the time-averaged queue length: $\theta = \left(\frac{1}{T}\right) \int_0^t N * Q(s)$ ds This is implemented by accumulating the area under the queue length curve.

```
import numpy as np

lam = 1.0
shape, rate = 3, 4
T_end = 100000

time = 0.0
queueLength = 0
server_busy = False

next_arrival = np.random.exponential(1 / lam)
next_departure = np.inf  # no departure is scheduled yet

area_queue = 0.0
last_event_time = 0.0

while time < T_end:
    nextEvent = min(next_arrival, next_departure)

    area_queue += queueLength * (nextEvent - last_event_time)
    last_event_time = nextEvent
    time = nextEvent

    if nextEvent == next_arrival: #arrival is next
        next_arrival = time + np.random.exponential(1 / lam)

        if not server_busy:
            server_busy = True
            service_time = np.random.gamma(shape, 1 / rate)
            next_departure = time + service_time
        else:
            queueLength += 1

    else:  # departure is next
        if queueLength > 0:
            queueLength -= 1
            service_time = np.random.gamma(shape, 1 / rate)
            next_departure = time + service_time
        else:
            server_busy = False
            next_departure = np.inf


theta_est = area_queue / T_end
print(f"Estimated θ ≈ {theta_est:.4f}")
```

```
rho = lam * (shape / rate)
E_S2 = shape * (shape + 1) / rate**2
theta_theoretical = lam**2 * E_S2 / (2 * (1 - rho))
print(f"Theoretical θ = {theta_theoretical:.4f}")
```

Problem 3a)

The state is $\{W_n\}$, the waiting time of customer n (not V_n). This represents the workload in the system just before customer n enters service.

The filtration $\mathcal{F}_n = \sigma(W_1, S_1, A_1, W_2, S_2, A_2, ..., W_n, S_n, A_n)$ contains all information up to and including the nth customer's arrival and service time. This makes $\{W_n\}$ a Markov process because:

Given W_n (current waiting time), the next state W_{n+1} depends only on W_n and the new i.i.d. pair (A_{n+1}, S_{n+1}) via Lindley's equation:

$$W_{n+1} = \max(0, W_n + S_n - A_{n+1})$$

The future evolution is independent of the past history $W_1, W_2, ..., W_{n-1}$ given the current state W_n
The state space is continuous: $W_n \in [0, \infty)$

Problem 3b)

Instead of tracking queue length over continuous time, we simulate the sojourn time $X_n = W_n + S_n$ for each customer n using Lindley's recursion:

- $W_{n+1} = \max(0, W_n + S_n - A_{n+1})$
- $X_n = W_n + S_n$

Procedure:

1. Initialize $W_0 = 0$ (system starts empty)
2. For $n = 1, 2, ..., N$:
   - Generate inter-arrival time $A_n \sim \text{Exp}(\lambda)$ and service time $S_n \sim \Gamma(3, 4)$
   - Compute $W_{n+1} = \max(0, W_n + S_n - A_n)$
   - Record sojourn time $X_n = W_n + S_n$
3. Estimate the stationary mean sojourn time: $\overline{X} \approx \frac{1}{N} \sum_{n=1}^{N} X_n$
4. Use Little's Law to obtain θ: since $\theta + \rho = \lambda \overline{X}$, we have

$$\hat{\theta} = \lambda \overline{X} - \rho$$

This Petri Net/Lindley approach is often more efficient computationally since we only process events at customer departures rather than tracking continuous-time queue dynamics.o covert these averaged soujourn times into the length of the average queue.