

# Queueing Simulation (Assignment 3) - GROUP 15

Miguel Ibrahim, Gadir Suleymanli, Timo Tromp , Lothar Arnou

December 17, 2023

## 1 Recapitulation of the model in assignment S-2, now with a battery

- a) To simulate the given scenario and determine the minimal battery capacities for the provided alpha values, we have come up with the following solution in Python (python files also provided in the submission):

```
1 import numpy as np
2 import simpy
3 import matplotlib.pyplot as plt
4 import copy
5 import random
6
7 import numpy as np
8
9 def generate_exponential_times(starttime, endtime, rate,
10                               num_samples):
11
12     values = np.random.exponential(scale=1/rate, size=
13                                     num_samples)
14
15     waardes = list(values)
16
17     times = []
18
19     current = 0
20     for i in range(0, len(waardes)):
21         temp = current + waardes[i]
22         times.append(temp)
23         current = temp
24
25     times = list(times)
26
27     if not all(starttime <= time <= endtime for time in times):
28         return generate_exponential_times(starttime, endtime,
29                                             rate, num_samples)
30     return times
31
32 class Gillespie:
33
34     def __init__(self, p01, p10):
35         self.p01 = p01
36         self.p10 = p10
```

```

37     def calculate_total_rate(self):
38         return self.p01 + self.p10
39
40     def Generate(self):
41         rando = random.uniform(0, 1)
42
43         if (rando < self.p01/self.calculate_total_rate()):
44             return 1
45         else:
46             return 0
47
48 class Singleton:
49
50     def __init__(self):
51         self.st = None
52
53     def singleton(self, list):
54         if self.st == None:
55             self.st = list
56             return self.st
57         else:
58             return self.st
59
60 class Process:
61
62     def __init__(self, alphas, gamma, l, p, e, gp = Gillespie
63 (0.5,0.5)):
64         self.alphas = alphas
65         self.gamma = gamma
66         self.p = p
67         self.e = e
68         self.gp = gp
69         self.l = l
70         self.total_energy_acquired = 1
71         self.total_energy_bought = 1
72         self.B = 0.1
73         self.results = []
74         self.amount_of_times = 0
75         self.output = None
76         self.jobs = []
77         self.done = False
78
79     def getOutput(self):
80
81         print("
82 -----")
83         print("Simulation Succesfull, running the simulation {
84 self.amount_of_times} the values are as follows:")
85         for alpha in self.alphas:
86             print(f"Alpha: {alpha}, Average Min Battery
87 Capacity: {self.results[self.alphas.index(alpha)] / self.
88 amount_of_times}")
89
90     def simulate(self, max_capacity, lambda_rate, min_capacity
91 = 1000, amount_of_times = 1, distr_value = 200 ,
92 capacity_increment = 10, time = 100000):
93
94         results = []
95         self.amount_of_times = amount_of_times
96         for alpha in self.alphas:

```

```

90         results.append(max_capacity)
91
92         # Run the simulations amount_of_times to get a more
average result
93         for i in range(1, amount_of_times+1):
94
95             arrivals = np.random.poisson(lam=lambda_rate)
96
97             arrival_times = generate_exponential_times(0, time,
self.l, arrivals)
98
99             jobs = []
100             original_wh = []
101
102             for j in range(0, len(arrival_times)):
103                 job = []
104                 tau_k = np.random.exponential(distr_value)
105                 W_k_new = np.random.exponential(distr_value)
106                 Wh = W_k_new*tau_k
107                 original_wh.append(Wh)
108                 job = [arrival_times[j], tau_k, Wh]
109                 jobs.append(job)
110
111             self.jobs = tuple(jobs)
112
113
114             self.output = []
115             for i in range(0, len(self.alphas)):
116                 self.output.append(max_capacity)
117
118             single = Singleton()
119
120             # For every alpha value do the simulation
121             for alpha in self.alphas:
122
123                 # Find a capacity by iterating over possible
values for it
124                 for capacity in range(max_capacity,
min_capacity, -capacity_increment):
125
126                     taak = single.singleton(copy.deepcopy(jobs)
)
127
128                     self.done = False
129
130
131                     env = simpy.Environment()
132
133                     # Run the simulation
134                     env.process(self.energy_source_process(env,
alpha, self.gamma, self.p, self.e, taak, capacity, self.
done))
135
136                     # Adjust simulation time accordingly
137                     env.run(until=time)
138
139                     if (self.done == True):
140                         #results[self.alphas.index(alpha)] +=
capacity

```

```

141         if (self.output[self.alphas.index(alpha)
142         )] > capacity):
143             self.output[self.alphas.index(alpha)
144             )] = capacity
145             print(f"using alpha: {alpha}, job {
146             self.alphas.index(alpha)} succesful with current min: {self
147             .output[self.alphas.index(alpha)]}")
148         else:
149             print(f"using alpha: {alpha}, job {self
150             .alphas.index(alpha)} failed when attempting capacity: {
151             capacity}, min is {self.output[self.alphas.index(alpha)]}")
152             print("Final", sorted(taak, key=lambda
153             x: x[2], reverse=True))
154             break
155         print("
156         -----")
157         print(f"Simulation Succesfull, running the simulation {
158         self.amount_of_times} times, the values are as follows:")
159         for alpha in self.alphas:
160             print(f"Alpha: {alpha}, Average Min Battery
161             Capacity: {self.output[self.alphas.index(alpha)] / self.
162             amount_of_times}")
163
164     def Execute(self, state, job, capacity, alpha, gamma, p, e)
165     :
166
167         res = None
168         finished = False
169         left = 0
170
171         if state == 1:
172             total_source = np.random.uniform(0.01, 2)
173             self.total_energy_acquired += total_source
174             # Consume energy from the intermittent source
175             if job[2] - total_source < 0: # Source is
176             enough to satisfy the job
177                 remaining_source = total_source - job[2]
178                 job[2] = 0
179                 if self.B + remaining_source > capacity: #
180                 Excess source cannot be stored
181                     self.B = capacity
182                     finished = True
183                     return res, finished, left
184                 else:
185                     self.B += e * remaining_source
186                     finished = True
187                     return res, finished, left
188             else: # Source is not enough, we need to check
189             the battery
190                 job[2] = job[2] - total_source
191                 if self.B - job[2] > 0: # Battery is enough
192                     self.B = self.B - job[2]
193                     job[2] = 0
194                     finished = True
195                     res = job[1]
196                     return res, finished, left
197                 else: # Battery is not enough
198                     job[2] = job[2] - self.B
199                     self.B = 0

```

```

186         left = job[2]
187         res = job[1]
188         return res, finished, left
189     # Check if the energy source is ON
190 else:
191     # Energy source is OFF
192     # Determine whether to reschedule or buy external
energy
193     if self.B - job[2] > 0: # Battery is enough
194         self.B = self.B - job[2]
195         job[2] = 0
196         finished = True
197         return res, finished, left
198     else: # Battery is not enough
199         job[2] = job[2] - self.B
200         self.B = 0
201         left = job[2]
202         if ((job[2] + self.total_energy_bought) / self.
total_energy_acquired <= alpha): #Buy external energy
203             cost = p * job[2]
204             self.total_energy_acquired += job[2]
205             self.total_energy_bought += job[2]
206             job[2] = 0
207             finished = True
208             return res, finished, left
209         else:
210             res = gamma
211             return res, finished, left
212
213
214
215 def energy_source_process(self, env, alpha, gamma, p, e,
tasks, capacity, done):
216
217     queue = []
218     self.total_energy_bought = 1
219     self.total_energy_acquired = 1
220
221     jobs = copy.deepcopy(tasks)
222
223     while len(jobs) > 0 or len(queue) > 0:
224
225
226         if (len(queue) > 0):
227
228             job = queue[0]
229
230             yield env.timeout(job[1])
231
232             state = self.gp.Generate()
233             logic, finished, left = self.Execute(state, job
, capacity, alpha, gamma, p, e)
234
235             if finished:
236                 queue.pop(0)
237             elif (logic == gamma):
238                 rs_time = np.random.exponential(gamma)
239                 new_job = [rs_time+job[0], job[1], left]
240                 queue.pop(0)
241                 if (rs_time+job[0] > env.now):

```

```

242         queue.append(new_job)
243     else:
244         jobs.append(new_job)
245         jobs = sorted(jobs, key=lambda x: x[0])
246     elif (logic != None):
247         yield env.timeout(logic)
248     else:
249         continue
250 else:
251     #wait untill time of execution
252     if (len(jobs) > 1):
253         for j in range(1, len(jobs)):
254             if (j >= len(jobs)):
255                 break
256             if (jobs[j][0] < jobs[0][0]):
257                 queue.append(jobs[j])
258                 jobs.pop(j)
259             else:
260                 break
261
262     yield env.timeout(jobs[0][0])
263
264     if (len(jobs) > 1):
265         if (j >= len(jobs)):
266             break
267         for j in range(1, len(jobs)):
268             if (j >= len(jobs)):
269                 break
270             if (jobs[j][0] < jobs[0][0]):
271                 queue.append(jobs[j])
272                 jobs.pop(j)
273             else:
274                 break
275     yield env.timeout(jobs[0][1])
276
277     #execute the job logic
278     state = self.gp.Generate()
279     logic, finished, left = self.Execute(state,
280     jobs[0] , capacity, alpha, gamma, p, e)
281
282     if (finished):
283         jobs.pop(0)
284     elif (logic == gamma):
285         rs_time = np.random.exponential(gamma)
286         new_job = [rs_time+jobs[0][0], jobs[0][1],
287         left]
288
289         jobs.pop(0)
290         jobs.append(new_job)
291         jobs = sorted(jobs, key=lambda x: x[0])
292     elif (logic != None):
293         if (len(jobs) > 1):
294             for j in range(1, len(jobs)):
295                 if (j >= len(jobs)):
296                     break
297                 if (jobs[j][0] < logic):
298                     queue.append(jobs[j])
299                     jobs.pop(j)
300                 else:
301                     break
302         yield env.timeout(logic)

```

```

300
301         if (jobs == []):
302             self.done = True

```

The above code initially generates multiple jobs that arrive at different times according to a Poisson distribution, and here we can apply the common variables method. We can do this by using the same job times (samples) for every configuration of the simulations. During the simulation,  $\tau$  and  $W_k$  values are generated for each job to be completed within the simulation.  $\tau$  refers to the time each process needs to complete, and  $W_k$  refers to the amount of watts the job needs during the completion time. By multiplying these variables we get the amount of Wh (watt-hour) that the job needs so that the system can use a similar unit of measurement. Then the simulation runs multiple times for each alpha value, as well as a wide range of capacity values, each incremented by at each iteration, until a minimal value is found that satisfies the given condition of running autonomously.

We have defined two states, ON (1) and OFF (0), as described in the task. The process switches between these two states using the Gillespie algorithm (line 8). We have tried to use reasonable values such as  $\lambda$  (rate of arrivals),  $e$  (Efficiency), and  $\gamma$  (rescheduling variable), which also resulted in reasonable values for the minimal capacities. We can simulate multiple times (line 151) for each scenario and then take the average of those simulations for every alpha value to ensure that the different variables for all the jobs generated do not affect the results significantly, and we can get some clear trends.

Within the function *energy\_source\_process*, we have many different steps to ensure that the user can run autonomously. Initially, we start with the job at the top of the queue. If the source is available, then we generate a certain amount of source that could be used to satisfy the requirements of the job, this generation of source uses a uniform distribution. If the job wattage can be satisfied with this amount, then the job is finished and it is removed from the queue. Otherwise, we check our battery storage at that given point and use the energy in the battery. If the battery does not have sufficient energy either, then we check if we can buy external energy, while still staying under the alpha value. If this is also not possible then the job remains in the queue, to see if it could be completed later on by generating more energy.

We assumed the  $\alpha$  to be the threshold value that represents the upper limit on the fraction of total energy demands that the system is allowed to satisfy by buying energy from external sources. So, for instance,  $\alpha = 0.01$  allows up to 1% of the total energy demands to be met by external purchases. In that sense, we kept track of all the energy that has been bought previously, as well as the total energy demands to see if the energy we are requiring is within the given threshold. If it is, we can safely buy the external energy. Otherwise, we rescheduled the job using the  $\gamma$  variable.

The obtained results after the simulation are shown in Figure 1.

The output in Figure 1 shows that, on average, the minimal capacity decreases as alpha increases.

```
Simulation Finished, amount of simulations ran: 24
Alpha: 0.0, Average Min Battery Capacity: 549.1666666666666
Alpha: 0.01, Average Min Battery Capacity: 528.3333333333334
Alpha: 0.05, Average Min Battery Capacity: 339.5833333333333
Alpha: 0.1, Average Min Battery Capacity: 148.33333333333334
```

Figure 1: Output of the Simulation (Single Run)

```
b) # Code same as part a, above
2     def plot_results(self, correlation_values):
3         min_battery_capacities = []
4
5         for alpha in self.alphas:
6             avg_min_capacity = 0
7
8             for correlation in correlation_values:
9                 my_time = 0
10                job_times = []
11
12                while my_time < 20000:
13                    new_job_time = np.random.poisson(lam=
lambda_rate) * 100
14                    my_time += new_job_time
15                    if my_time < 20000:
16                        job_times.append(my_time)
17
18                results = []
19
20                for capacity in range(0, 2001, 10):
21                    job_times_copy = copy.copy(job_times)
22                    env = simpy.Environment()
23
24                    energy_source = env.process(self.
energy_source_process(env, alpha, self.gamma, self.p, self.
e, job_times_copy, capacity, correlation))
25
26                    env.run(until=20000)
27
28                    if not job_times_copy:
29                        results.append(capacity)
30                        break
31                    else:
32                        continue
33
34                if results: # Only calculate the average when
there are results
35                    avg_min_capacity += sum(results) / len(
results)
36
37                if avg_min_capacity != 0:
38                    avg_min_capacity /= len(correlation_values)
39                min_battery_capacities.append(avg_min_capacity)
40
41            plt.plot(self.alphas, min_battery_capacities, marker='o
',)
42            plt.xlabel('Alpha')
43            plt.ylabel('Average Min Battery Capacity')
44            plt.title('Influence of Correlation on System
Performance')
45            plt.grid(True)
```



```

46         plt.show()
47
48 if __name__ == "__main__":
49     lambda_rate = 0.2
50     gamma = 0.5
51     p = 0.2
52     e = 0.8
53
54     p = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
55     correlation_values = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0] #
56     Different correlation levels
57     p.plot_results(correlation_values)

```

The above code is an extension of part a, it adds a plotting mechanism (def plot\_results(self, correlation\_values)) to view the correlation factor of the ON/OFF process as the program simulates the environment. The results are plotted to assess how different correlations in the ON-OFF process affect the system's performance under various conditions.

The output of part b can be found in Figure 2. This figure shows the correlation between the increased allowed import of energy and the battery capacity. In the graph, battery capacity decreases as alpha increases. This makes sense because increasing the capacity of the battery translates to being able to hold up more energy that might be needed later. It also makes sense because alpha determines how much external energy can be purchased, which allows the user to run autonomously without requiring as much minimal battery capacity. When additional resources are needed, they can be purchased. Now we rely on another external system to aid the current system in its path towards running autonomously, therefore not requiring as much battery capacity as required when the system is fully autonomous (When  $\alpha = 0$ ).

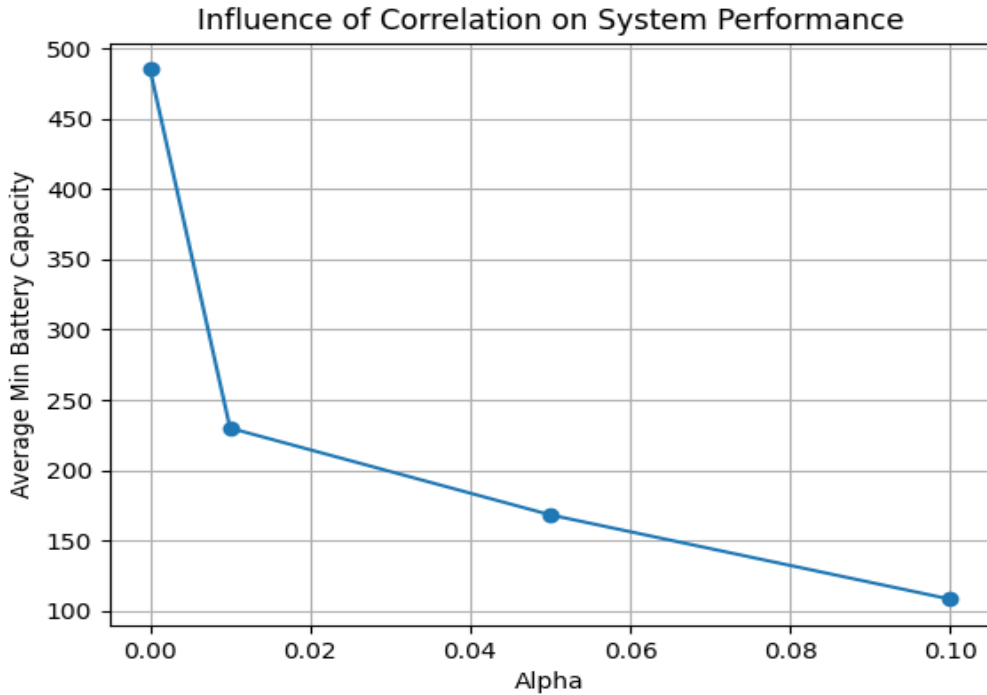


Figure 2: Influence of Correlation on System Performance

## 2 Scenarios with N users

```
a) import numpy as np
2 import simpy
3 import matplotlib.pyplot as plt
4 import copy
5 import random
6 import threading
7
8 from concurrent.futures import ThreadPoolExecutor, as_completed
9
10 class Gillespie(object):
11
12     def __init__(self, p01, p10):
13         self.p01 = p01
14         self.p10 = p10
15
16     def calculate_total_rate(self):
17         return self.p01 + self.p10
18
19     def Generate(self):
20         rando = random.uniform(0, 1)
21         if (rando < self.p01 / self.calculate_total_rate()):
22             return 1
23         else:
24             return 0
25
26     def setP01(self, p):
27         self.p01 = p
28
29     def setP10(self, p):
30         self.p10 = p
31
32
33 class ExclusiveGillespie(Gillespie):
34
35     def __init__(self, p01, p10):
36         super().__init__(p01, p10)
37         self.first = True
38         self.lock = threading.Lock()
39         self.current = None
40
41     def Generate(self):
42         with self.lock:
43             if (self.first == True):
44                 self.first = False
45                 self.current = super().Generate()
46             return self.current
47
48     def next(self):
49         with self.lock:
50             self.first = True
51
52     def setP01(self, p):
53         with self.lock:
54             return super().setP01(p)
55
56     def setP10(self, p):
57         with self.lock:
58             return super().setP10(p)
```

```

59
60 class Energy_Collector:
61
62     def __init__(self):
63         self.energy = 0
64         self.total_additions = 0
65         self.total_usage = 0
66         self.lock = threading.Lock()
67
68     def getEnergy(self):
69         with self.lock:
70             return self.energy
71
72     def addEnergy(self, value):
73         with self.lock:
74             self.energy += value
75             self.total_additions += value
76
77     def useEnergy(self, value):
78         with self.lock:
79             self.energy -= value
80             self.total_usage -= value
81
82     def Check(self):
83         #debugging
84         if (round(self.getEnergy(), 5) == round(self.
85 getTotalAddition() + self.getTotalUsage(), 5)):
86             s = 'True, final values are: Remaining Energy: {},
87 Total Balnce: {}, Total Addition: {}, Total Usage: {}'.
88 format(round(self.getEnergy(), 5), round(self.
89 getTotalAddition() + self.getTotalUsage(), 5) ,round(self.
90 getTotalAddition(), 5), round(self.getTotalUsage(),5))
91             print(s)
92             return 1
93         else:
94             s = 'False, final values are: Remaining Energy: {},
95 Total Balnce: {}, Total Addition: {}, Total Usage: {}'.
96 format(round(self.getEnergy(), 5), round(self.
97 getTotalAddition() + self.getTotalUsage(), 5) ,round(self.
98 getTotalAddition(), 5), round(self.getTotalUsage(),5))
99             print(s)
100             return 0
101
102     def getTotalUsage(self):
103         #debugging
104         with self.lock:
105             return self.total_usage
106
107     def getTotalAddition(self):
108         #debugging
109         with self.lock:
110             return self.total_additions
111
112 class Simulator:
113     def __init__(self, max_lambda=1.0, lambda_step=0.1):
114         self.processes = []
115         self.Battery = Energy_Collector()
116         self.max_lambda = max_lambda
117         self.lambda_step = lambda_step
118         self.lambdas = []

```

```

110
111     def addUser(self, process):
112         process.setBattery(self.Battery)  # Pass the Battery
113         object to the process
114         self.processes.append(process)
115
116     def deleteProcess(self, process):
117         self.processes.remove(process)
118
119     def readLambdas(self):
120         for i in self.lambdas:
121             print(i)
122
123     def simulate(self, max_capacity=2001, min_capacity = 0,
124         amount_of_times = 1, capacity_increment = 10, time = 20000)
125         :
126         #Function to simulate different processes per user
127         #need to multiproces this function
128         #scenario ii
129
130         max_lambda = self.max_lambda
131         lambda_step = self.lambda_step
132         while max_lambda > 0:
133             for user in self.processes:
134                 user.gp.setP01(max_lambda)
135                 user.gp.setP10(max_lambda)
136                 with ThreadPoolExecutor(max_workers=4) as executor:
137                     futures = [executor.submit(user.simulate,
138 max_capacity, min_capacity, amount_of_times,
139 capacity_increment, time) for user in self.processes]
140                     for future in futures:
141                         future.result()
142                     backlog = False
143                     for user in self.processes:
144                         if user.getResults()[0] > 0:
145                             backlog = True
146                             break
147                     if backlog:
148                         max_lambda -= lambda_step
149                         break
150                     else:
151                         max_lambda += lambda_step
152                         self.Battery.Check()
153                         self.lambdas.append(max_lambda)
154
155         for process in self.processes:
156             process.getOutput()
157
158     def simulate_all_same_on_off_process(self, gp =
159 ExclusiveGillespie(0.5, 0.5), max_capacity=2001,
160 min_capacity = 0, amount_of_times = 1, capacity_increment =
161 10, time = 20000):
162         #scenario i
163         for process in self.processes:
164             process.setGP(gp)
165             self.simulate(max_capacity, min_capacity,
166 amount_of_times, capacity_increment, time)
167
168

```

```

160     def find_max_lambda(self, max_capacity=2001, min_capacity
161     =0, amount_of_times=1, capacity_increment=10, time=20000):
162         max_lambda = self.max_lambda
163         lambda_step = self.lambda_step
164         while max_lambda > 0:
165             for user in self.processes:
166                 user.gp.setP01(max_lambda)
167                 user.gp.setP10(max_lambda)
168                 with ThreadPoolExecutor(max_workers=len(self.
169 processes)) as executor:
170                     futures = [executor.submit(user.simulate,
171 max_capacity, min_capacity, amount_of_times,
172 capacity_increment, time) for user in self.processes]
173                     for future in futures:
174                         future.result()
175                     backlog = False
176                     for user in self.processes:
177                         if user.getResults()[0] > 0:
178                             backlog = True
179                             break
180                     if backlog:
181                         max_lambda -= lambda_step
182                         break
183                     else:
184                         max_lambda += lambda_step
185         return max_lambda
186
187 class Process:
188
189     def __init__(self, alphas, gamma, p, e, gp = Gillespie(0.5,
190 0.5)):
191         self.alphas = alphas
192         self.gamma = gamma
193         self.p = p
194         self.e = e
195         self.gp = gp
196         self.Battery = None
197         self.results = None
198         self.amount_of_times = 0
199
200     def getResults(self):
201         if (self.results == None):
202             return "Simulate the process first"
203         else:
204             return self.results
205
206     def setBattery(self, battery):
207         self.Battery = battery
208
209     def setGP(self, gp):
210         self.gp = gp
211
212     def energy_source_process(self, env, alpha, gamma, p, e,
213 job_times, capacity):
214
215         state = self.gp.Generate()
216         total_energy_demand = 0
217         B = 0.1
218         total_energy_bought = 0
219         jobs = []

```

```

214         my_time = 0
215
216         while job_times != []:
217             # Generate a new job
218             state = self.gp.Generate()
219
220             if job_times and job_times[0] <= my_time:
221                 tau_k = np.random.exponential(lambda_rate)
222                 W_k_new = np.random.exponential(2000) # Random
223                 wattage for the job
224                 Wh_new = tau_k * W_k_new
225                 job_times.pop(0)
226                 total_energy_demand += Wh_new
227                 jobs.append(Wh_new)
228                 my_time += 10
229
230             if jobs == []:
231                 continue
232             if state == 1:
233                 total_source = np.random.uniform(0, 10)
234                 # Consume energy from the intermittent source
235                 if jobs[0] - total_source < 0: # Source is
236                     enough to satisfy the job
237                     remaining_source = total_source - jobs[0]
238                     jobs[0] = 0
239                     jobs.pop(0)
240                     if B + remaining_source > capacity: #
241                         Excess source cannot be stored
242                         self.Battery.addEnergy(remaining_source
243 )
244                         B = capacity
245                     else:
246                         B = B + e * remaining_source
247                     else: # Source is not enough, we need to check
248                         the battery
249                         jobs[0] = jobs[0] - total_source
250                         if B - jobs[0] > 0: # Battery is enough
251                             B = B - jobs[0]
252                             jobs[0] = 0
253                             jobs.pop(0)
254                             yield env.timeout(1 / tau_k)
255
256                         else: # Battery is not enough
257                             jobs[0] = jobs[0] - B
258                             B = 0
259                             yield env.timeout(1 / tau_k)
260                         # Check if the energy source is ON
261                         else:
262                             # Energy source is OFF
263                             # Determine whether to reschedule or buy
264                             external energy
265                             if B - jobs[0] > 0: # Battery is enough
266                                 B = B - jobs[0]
267                                 jobs[0] = 0
268                                 jobs.pop(0)
269                             else: # Battery is not enough
270                                 jobs[0] = jobs[0] - B
271                                 B = 0
272                                 if (((jobs[0] + total_energy_bought) /
273 total_energy_demand <= alpha) and (self.Battery.getEnergy()

```

```

267         > jobs[0])): #Buy external energy
268             cost = p * jobs[0]
269             total_energy_bought += jobs[0]
270             self.Battery.useEnergy(jobs[0])
271             jobs[0] = 0
272         else:
273             yield env.timeout(np.random.exponential
274 (gamma)) # Reschedule
275             #print(test)
276
277     def simulate(self, max_capacity= 2000, min_capacity=0,
278 amount_of_times=1, capacity_increment=10, time=20000):
279         results = []
280         self.amount_of_times=amount_of_times
281         for alpha in self.alphas:
282             results.append(0)
283
284             for i in range(1, amount_of_times + 1):
285                 for alpha in self.alphas:
286                     my_time = 0
287                     job_times = []
288
289                     while my_time < time:
290                         new_job_time = np.random.poisson(lam=
291 lambda_rate) * 100
292                         my_time += new_job_time
293                         if my_time < time:
294                             job_times.append(my_time)
295
296                     for capacity in list(range(min_capacity,
297 max_capacity, capacity_increment)):
298                         job_times_copy = copy.copy(job_times)
299                         env = simpy.Environment()
300                         env.process(self.energy_source_process(env,
301 alpha, self.gamma, self.p, self.e, job_times_copy,
302 capacity))
303                         env.run(until=time) # Adjust simulation
304 time accordingly
305
306                     if not job_times_copy:
307                         results[self.alphas.index(alpha)] +=
308 capacity
309                     break # Move on to the next alpha
310                 else:
311                     continue
312             if (isinstance(self.gp, ExclusiveGillespie)):
313                 self.gp.next()
314
315         self.results = results
316         #self.getOutput()
317
318     def getOutput(self):
319
320         if (self.Battery.Check()):
321             print("
322 -----")
323             print("Simulation Succesfull, running the
324 simulation {self.amount_of_times} the values are as follows
325 :")

```

```

315         for alpha in self.alphas:
316             print(f"Alpha: {alpha}, Average Min Battery
Capacity: {self.results[self.alphas.index(alpha)] / self.
amount_of_times}")
317
318 lambda_rate = 0.2 # Poisson stream rate
319 gamma = 0.5 # Rescheduling rate
320 p = 0.2 # External energy purchase rate
321 e = 0.8 # Battery charging efficiency
322 B_values = [] # To store minimal battery capacities for
different alpha values
323
324 # Create User instances
325 user1 = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
326 user2 = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
327 user3 = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
328
329 # Initialize the Simulator
330 s = Simulator(max_lambda=1.0, lambda_step=0.01)
331
332 # Add User instances to the simulator
333 s.addUser(user1)
334 s.addUser(user2)
335 s.addUser(user3)
336
337 #s.simulate()
338 s.simulate_all_same_on_off_process(amount_of_times=10)
339
340 print(s.readLambdas())

```

This Python code simulates a scenario involving multiple users equipped with batteries and varying energy demands. Each user's behavior is represented as a Markov On-Off process. The different scenarios differ in that if the general state is zero in the first scenario a situation can occur where can able to be bought for a while. This might lead to a higher dependency on chance compared to the second scenario. In the second scenario, every process can generate it's own state, since there are many users the low of averages dictates that there will always be a pool of users contributing to the energy pool. This makes the second scenario more stable and consistent than the first scenario. So we can expect that  $\lambda_{max}$  will on average be bigger in scenario 2 compared to scenario 1, our simulation results also supported this assumption. We implemented the logic for the scenario 1 using the *simulate\_all\_same\_on\_off\_process()* and for scenario 2, using *simulate()* functions. We then extracted the values using our function for  $\lambda$  calculations.

- b) If we were to use the regeneration method in scenario ii, then let's assume that a regeneration cycle for each user is  $X_i(t)$ . We can consider the regeneration period to start when the process turns to 'ON' state, and ends when the process returns to 'OFF' state. Then, for each user, we would have a specific duration for the regeneration cycle,  $t_i$ . Our expected regeneration period for a single user would then be:

$$E[t_1] = \frac{1}{\lambda} \quad (1)$$

In order to obtain the lower bound, we would have to look at the minimum among all the users. However, the duration of regeneration cycle,  $t_i$  is identi-



cally distributed, so the minimum of them would become:

$$E[t_{min}] = \min_{i=1}^N E[t_i] = E[t_1] = \frac{1}{\lambda} \quad (2)$$

As a result, the lower bound of the expected regeneration period does not depend on the number of users  $N$  for scenario ii. Regardless of the number of users, the system regenerates at a rate determined by the underlying Poisson stream of jobs. In this case, the regeneration method could be useful in simplifying the analysis, and the number of users will not affect it. So, scaling the process to have more users will not complicate the application of regeneration method in this particular case, since expected regeneration period does not increase with the number of users.

A control variate is a statistical technique used in experimental design and data analysis to reduce the variance (i.e., the spread or variability) of the estimates or outcomes of an experiment or simulation. It is a method for improving the precision and efficiency of statistical estimates or reducing the standard errors of those estimates. We used control variate in this case to create the following code for part a

### 3 A spatial, time-evolving process

a) The following is our attempt at Part 3:

```

1  # Import additional libraries
2  from scipy.spatial.distance import pdist, squareform
3
4  import numpy as np
5  import simpy
6  import matplotlib.pyplot as plt
7  import copy
8  import random
9  import threading
10
11 from concurrent.futures import ThreadPoolExecutor, as_completed
12
13 class Gillespie(object):
14
15     def __init__(self, p01, p10):
16         self.p01 = p01
17         self.p10 = p10
18
19     def calculate_total_rate(self):
20         return self.p01 + self.p10
21
22     def Generate(self):
23         rando = random.uniform(0, 1)
24         if (rando < self.p01 / self.calculate_total_rate()):
25             return 1
26         else:
27             return 0
28
29     def setP01(self, p):
30         self.p01 = p
31
32     def setP10(self, p):
33         self.p10 = p

```

```

34
35
36 class ExclusiveGillespie(Gillespie):
37
38     def __init__(self, p01, p10):
39         super().__init__(p01, p10)
40         self.first = True
41         self.lock = threading.Lock()
42         self.current = None
43
44     def Generate(self):
45         with self.lock:
46             if (self.first == True):
47                 self.first = False
48                 self.current = super().Generate()
49             return self.current
50
51     def next(self):
52         with self.lock:
53             self.first = True
54
55     def setP01(self, p):
56         with self.lock:
57             return super().setP01(p)
58
59     def setP10(self, p):
60         with self.lock:
61             return super().setP10(p)
62
63 class Energy_Collector:
64
65     def __init__(self):
66         self.energy = 0
67         self.total_additions = 0
68         self.total_usage = 0
69         self.lock = threading.Lock()
70
71     def getEnergy(self):
72         with self.lock:
73             return self.energy
74
75     def addEnergy(self, value):
76         with self.lock:
77             self.energy += value
78             self.total_additions += value
79
80     def useEnergy(self, value):
81         with self.lock:
82             self.energy -= value
83             self.total_usage -= value
84
85     def Check(self):
86         #debugging
87         if (round(self.getEnergy(), 5) == round(self.
88             getTotalAddition() + self.getTotalUsage(), 5)):
89             s = 'True, final values are: Remaining Energy: {},
90 Total Balnce: {}, Total Addition: {}, Total Usage: {}'.
91             format(round(self.getEnergy(), 5), round(self.
92                 getTotalAddition() + self.getTotalUsage(), 5), round(self.
93                     getTotalAddition(), 5), round(self.
94                         getTotalUsage(), 5))

```

```

89         print(s)
90         return 1
91     else:
92         s = 'False, final values are: Remaining Energy: {},
Total Balnce: {}, Total Addition: {}, Total Usage: {}'.
format(round(self.getEnergy(), 5), round(self.
getTotalAddition() + self.getTotalUsage(), 5) ,round(self.
getTotalAddition(), 5), round(self.getTotalUsage(),5))
93         print(s)
94         return 0
95
96     def getTotalUsage(self):
97         #debugging
98         with self.lock:
99             return self.total_usage
100
101     def getTotalAddition(self):
102         #debugging
103         with self.lock:
104             return self.total_additions
105
106 class Simulator:
107     def __init__(self, max_lambda=1.0, lambda_step=0.1):
108         self.processes = []
109         self.Battery = Energy_Collector()
110         self.max_lambda = max_lambda
111         self.lambda_step = lambda_step
112         self.lambdas = []
113
114     def addUser(self, process):
115         process.setBattery(self.Battery) # Pass the Battery
object to the process
116         self.processes.append(process)
117
118     def deleteProcess(self, process):
119         self.processes.remove(process)
120
121     def readLambdas(self):
122         for i in self.lambdas:
123             print(i)
124
125     def simulate(self, max_capacity=2001, min_capacity = 0,
amount_of_times = 1, capacity_increment = 10, time = 20000)
:
126         #Function to simulate different processes per user
127         #need to multiproces this function
128         #scenario ii
129
130         max_lambda = self.max_lambda
131         lambda_step = self.lambda_step
132         while max_lambda > 0:
133             for user in self.processes:
134                 user.gp.setP01(max_lambda)
135                 user.gp.setP10(max_lambda)
136                 with ThreadPoolExecutor(max_workers=4) as executor:
137                     futures = [executor.submit(user.simulate,
max_capacity, min_capacity, amount_of_times,
capacity_increment, time) for user in self.processes]
138                     for future in futures:
139                         future.result()

```

```

140         backlog = False
141         for user in self.processes:
142             if user.getResults()[0] > 0:
143                 backlog = True
144                 break
145         if backlog:
146             max_lambda -= lambda_step
147             break
148         else:
149             max_lambda += lambda_step
150         self.Battery.Check()
151         self.lambdas.append(max_lambda)
152
153         for process in self.processes:
154             process.getOutput()
155
156
157     def simulate_all_same_on_off_process(self, gp =
ExclusiveGillespie(0.5, 0.5), max_capacity=2001,
min_capacity = 0, amount_of_times = 1, capacity_increment =
10, time = 20000):
158         #scenario i
159         for process in self.processes:
160             process.setGP(gp)
161             self.simulate(max_capacity, min_capacity,
amount_of_times, capacity_increment, time)
162
163     def find_max_lambda(self, max_capacity=2001, min_capacity
=0, amount_of_times=1, capacity_increment=10, time=20000):
164         max_lambda = self.max_lambda
165         lambda_step = self.lambda_step
166         while max_lambda > 0:
167             for user in self.processes:
168                 user.gp.setP01(max_lambda)
169                 user.gp.setP10(max_lambda)
170             with ThreadPoolExecutor(max_workers=len(self.
processes)) as executor:
171                 futures = [executor.submit(user.simulate,
max_capacity, min_capacity, amount_of_times,
capacity_increment, time) for user in self.processes]
172                 for future in futures:
173                     future.result()
174             backlog = False
175             for user in self.processes:
176                 if user.getResults()[0] > 0:
177                     backlog = True
178                     break
179             if backlog:
180                 max_lambda -= lambda_step
181                 break
182             else:
183                 max_lambda += lambda_step
184         return max_lambda
185
186 class Process:
187
188     def __init__(self, alphas, gamma, p, e, gp = Gillespie(0.5,
0.5)):
189         self.alphas = alphas
190         self.gamma = gamma

```

```

191         self.p = p
192         self.e = e
193         self.gp = gp
194         self.Battery = None
195         self.results = None
196         self.amount_of_times = 0
197
198     def getResults(self):
199         if (self.results == None):
200             return "Simulate the process first"
201         else:
202             return self.results
203
204     def setBattery(self, battery):
205         self.Battery = battery
206
207     def setGP(self, gp):
208         self.gp = gp
209
210     def energy_source_process(self, env, alpha, gamma, p, e,
211                               job_times, capacity):
212
213         state = self.gp.Generate()
214         total_energy_demand = 0
215         B = 0.1
216         total_energy_bought = 0
217         jobs = []
218         my_time = 0
219
220         while job_times != []:
221             # Generate a new job
222             state = self.gp.Generate()
223
224             if job_times and job_times[0] <= my_time:
225                 tau_k = np.random.exponential(lambda_rate)
226                 W_k_new = np.random.exponential(2000) # Random
227                 wattage for the job
228                 Wh_new = tau_k * W_k_new
229                 job_times.pop(0)
230                 total_energy_demand += Wh_new
231                 jobs.append(Wh_new)
232                 my_time += 10
233
234             if jobs == []:
235                 continue
236             if state == 1:
237                 total_source = np.random.uniform(0, 10)
238                 # Consume energy from the intermittent source
239                 if jobs[0] - total_source < 0: # Source is
240                     enough to satisfy the job
241                     remaining_source = total_source - jobs[0]
242                     jobs[0] = 0
243                     jobs.pop(0)
244                     if B + remaining_source > capacity: #
245                         Excess source cannot be stored
246                         self.Battery.addEnergy(remaining_source
247 )
248
249                     B = capacity
250             else:
251                 B = B + e * remaining_source

```

```

246         else: # Source is not enough, we need to check
the battery
247             jobs[0] = jobs[0] - total_source
248             if B - jobs[0] > 0: # Battery is enough
249                 B = B - jobs[0]
250                 jobs[0] = 0
251                 jobs.pop(0)
252                 yield env.timeout(1 / tau_k)
253
254             else: # Battery is not enough
255                 jobs[0] = jobs[0] - B
256                 B = 0
257                 yield env.timeout(1 / tau_k)
258         # Check if the energy source is ON
259         else:
260             # Energy source is OFF
261             # Determine whether to reschedule or buy
external energy
262             if B - jobs[0] > 0: # Battery is enough
263                 B = B - jobs[0]
264                 jobs[0] = 0
265                 jobs.pop(0)
266             else: # Battery is not enough
267                 jobs[0] = jobs[0] - B
268                 B = 0
269                 if (((jobs[0] + total_energy_bought) /
total_energy_demand <= alpha) and (self.Battery.getEnergy()
> jobs[0])): #Buy external energy
270                     cost = p * jobs[0]
271                     total_energy_bought += jobs[0]
272                     self.Battery.useEnergy(jobs[0])
273                     jobs[0] = 0
274                 else:
275                     yield env.timeout(np.random.exponential
(gamma)) # Reschedule
276                     #print(test)
277
278
279     def simulate(self, max_capacity= 2000, min_capacity=0,
amount_of_times=1, capacity_increment=10, time=20000):
280         results = []
281         self.amount_of_times=amount_of_times
282         for alpha in self.alphas:
283             results.append(0)
284
285         for i in range(1, amount_of_times + 1):
286             for alpha in self.alphas:
287                 my_time = 0
288                 job_times = []
289
290                 while my_time < time:
291                     new_job_time = np.random.poisson(lam=
lambda_rate) * 100
292                     my_time += new_job_time
293                     if my_time < time:
294                         job_times.append(my_time)
295
296                 for capacity in list(range(min_capacity,
max_capacity, capacity_increment)):
297                     job_times_copy = copy.copy(job_times)

```

```

298         env = simpy.Environment()
299         env.process(self.energy_source_process(env,
alpha, self.gamma, self.p, self.e, job_times_copy,
capacity))
300         env.run(until=time) # Adjust simulation
time accordingly
301
302         if not job_times_copy:
303             results[self.alphas.index(alpha)] +=
capacity
304             break # Move on to the next alpha
305         else:
306             continue
307         if (isinstance(self.gp, ExclusiveGillespie)):
308             self.gp.next()
309
310         self.results = results
311         #self.getOutput()
312
313     def getOutput(self):
314
315         if (self.Battery.Check()):
316             print("
-----")
317             print("Simulation Succesfull, running the
simulation {self.amount_of_times} the values are as follows
:")
318             for alpha in self.alphas:
319                 print(f"Alpha: {alpha}, Average Min Battery
Capacity: {self.results[self.alphas.index(alpha)] / self.
amount_of_times}")
320
321
322 class RingGraphSimulator(Simulator):
323     def __init__(self, num_users, ring_rate, alphas=None,
max_lambda=1.0, lambda_step=0.1, **kwargs):
324         super().__init__(max_lambda, lambda_step)
325         if alphas is None:
326             alphas = [] # Set a default value or handle it as
needed
327         self.alphas = alphas
328         self.num_users = num_users
329         self.ring_rate = ring_rate
330         self.distance_matrix = self.
generate_ring_distance_matrix()
331
332         self.processes = []
333         self.Battery = Energy_Collector()
334         self.gamma = gamma
335         self.p = p
336         self.e = e
337         self.lambdas = []
338
339     def generate_ring_distance_matrix(self):
340         positions = np.linspace(0, 2 * np.pi, self.num_users,
endpoint=False)
341         positions = np.column_stack([np.cos(positions), np.sin(
positions)])
342         return squareform(pdist(positions))
343

```

```

344 def generate_ring_correlated_states(self, env, user, alpha,
345 gamma, p, e, job_times, capacity):
346     state = user.gp.Generate()
347     total_energy_demand = 0
348     B = 0.1
349     total_energy_bought = 0
350     jobs = []
351     my_time = 0
352
353     while job_times:
354         # Generate a new job
355         state = user.gp.Generate()
356
357         if job_times and job_times[0] <= my_time:
358             tau_k = np.random.exponential(lambda_rate)
359             W_k_new = np.random.exponential(2000) # Random
360             wattage for the job
361             Wh_new = tau_k * W_k_new
362             job_times.pop(0)
363             total_energy_demand += Wh_new
364             jobs.append(Wh_new)
365             my_time += 10
366
367         if not jobs:
368             continue
369
370         if state == 1:
371             total_source = np.random.uniform(0, 10)
372             # Consume energy from the intermittent source
373             if jobs[0] - total_source < 0: # Source is
374                 enough to satisfy the job
375                 remaining_source = total_source - jobs[0]
376                 jobs[0] = 0
377                 jobs.pop(0)
378                 if B + remaining_source > capacity: #
379                     Excess source cannot be stored
380                     user.Battery.addEnergy(remaining_source
381 )
382                     B = capacity
383                 else:
384                     B = B + e * remaining_source
385             else: # Source is not enough, we need to check
386                 the battery
387                 jobs[0] = jobs[0] - total_source
388                 if B - jobs[0] > 0: # Battery is enough
389                     B = B - jobs[0]
390                     jobs[0] = 0
391                     jobs.pop(0)
392                     yield env.timeout(1 / tau_k)
393
394                 else: # Battery is not enough
395                     jobs[0] = jobs[0] - B
396                     B = 0
397                     yield env.timeout(1 / tau_k)
398
399         else:
400             # Energy source is OFF
401             # Determine whether to reschedule or buy
402             external energy
403             if B - jobs[0] > 0: # Battery is enough
404                 B = B - jobs[0]

```



```

397         jobs[0] = 0
398         jobs.pop(0)
399         else: # Battery is not enough
400             jobs[0] = jobs[0] - B
401             B = 0
402             if ((jobs[0] + total_energy_bought) /
total_energy_demand <= alpha) and (
403                 user.Battery.getEnergy() > jobs[0])
: # Buy external energy
404                 cost = p * jobs[0]
405                 total_energy_bought += jobs[0]
406                 user.Battery.useEnergy(jobs[0])
407                 jobs[0] = 0
408             else:
409                 yield env.timeout(np.random.exponential
(gamma)) # Reschedule
410
411     def simulate_ring_graph(self, gp=ExclusiveGillespie(0.5,
0.5), max_capacity=2001, min_capacity=0,
412                             amount_of_times=1,
capacity_increment=10, time=20000):
413         for process in self.processes:
414             process.setGP(gp)
415
416         for user in self.processes:
417             env = simpy.Environment()
418             user_processes = [self.
generate_ring_correlated_states(env, user, user.gp.p01,
self.gamma, self.p, self.e,
419
                                [], max_capacity) for _ in range(amount_of_times)]
420             env.process(self.ring_process(env, user_processes,
max_capacity, min_capacity, amount_of_times,
421                                         capacity_increment,
time))
422             env.run(until=time)
423
424     def ring_process(self, env, user_processes, max_capacity,
min_capacity, amount_of_times, capacity_increment, time):
425         for alpha in self.alphas:
426             results = []
427
428             for i in range(amount_of_times):
429                 user_jobs = [list(process) for process in
user_processes]
430
431                 my_time = 0
432                 job_times = []
433
434                 while my_time < time:
435                     my_time += 10
436                     yield env.timeout(1)
437                     new_job_time = np.random.poisson(lam=
lambda_rate) * 100
438                     my_time += new_job_time
439                     if my_time < time:
440                         job_times.append(my_time)
441
442                 for capacity in range(min_capacity,
max_capacity, capacity_increment):

```

```

443         job_times_copy = copy.copy(job_times)
444         user_jobs = [list(self.
generate_ring_correlated_states(env, self.processes[i],
alpha, self.gamma, self.p, self.e, job_times_copy, capacity
)) for i in range(len(self.processes))]
445
446         env.run(until=time) # Adjust simulation
time accordingly
447
448         if not job_times_copy:
449             results.append(sum([self.processes[i].
Battery.getEnergy() for i in range(len(self.processes))]))
450             break # Move on to the next alpha
451         else:
452             continue
453
454         average_result = sum(results) / len(results)
455         print(f"Alpha: {alpha}, Average Min Battery
Capacity: {average_result}")
456
457
458     def calculate_temporal_correlation(self, user_processes):
459         # Flatten the generator and convert it to a list
460         flat_processes = [item for sublist in user_processes
for item in sublist]
461
462         # Check if the flattened list is not empty
463         if flat_processes:
464             # Calculate temporal correlation for a single user
465             return np.corrcoef(flat_processes)
466         else:
467             print("User processes are empty.")
468             return np.empty((0, 0))
469
470     def calculate_spatial_correlation(self, user_results):
471         # Check if user_results is empty
472         if not user_results or not user_results[0]:
473             print("Spatial correlation data is empty.")
474             return np.empty((0, 0))
475
476         # Calculate spatial correlation between different users
477         return np.corrcoef(user_results)
478
479     def plot_temporal_correlation(self,
temporal_correlation_matrix):
480         # Flatten the matrix to a 1D array
481         if temporal_correlation_matrix.size > 0:
482             plt.imshow(temporal_correlation_matrix, cmap='
viridis', origin='lower', interpolation='none')
483             plt.colorbar()
484             plt.title('Temporal Correlation Matrix')
485             plt.show()
486         else:
487             print("Temporal correlation matrix is empty.")
488
489     def plot_spatial_correlation(self,
spatial_correlation_matrix):
490         # Ensure that the matrix is not empty
491         if spatial_correlation_matrix.size > 0:

```

```

492         plt.imshow(spatial_correlation_matrix, cmap='
viridis', origin='lower', interpolation='none')
493         plt.colorbar()
494         plt.title('Temporal Correlation Matrix')
495         plt.show()
496     else:
497         print("Temporal correlation matrix is empty.")
498
499
500     def analyze_correlations(self, gp=ExclusiveGillespie(0.5,
0.5), max_capacity=2001, min_capacity=0,
501                             amount_of_times=1, capacity_increment
=10, time=20000):
502         for process in self.processes:
503             process.setGP(gp)
504
505         user_processes = []
506
507         for user in self.processes:
508             env = simpy.Environment()
509             processes = [self.generate_ring_correlated_states(
env, user, user.gp.p01, self.gamma, self.p, self.e,
510
[], max_capacity) for _ in range(amount_of_times)]
511             env.process(self.ring_process(env, processes,
max_capacity, min_capacity, amount_of_times,
512
capacity_increment,
time))
513             env.run(until=time)
514             user_processes.append(processes)
515
516         # Analyze temporal correlation
517         temporal_correlation_matrix = self.
calculate_temporal_correlation(user_processes[0][0])
518         self.plot_temporal_correlation(
temporal_correlation_matrix)
519
520         # Analyze spatial correlation
521         user_results = [
522             [sum([self.processes[i].Battery.getEnergy() for i in
range(len(self.processes))]) for _ in range(amount_of_times
)]
523
524             for _ in range(len(self.processes))]
525         spatial_correlation_matrix = self.
calculate_spatial_correlation(user_results)
526         self.plot_spatial_correlation(
spatial_correlation_matrix)
527
528 # ...
529
530 lambda_rate = 0.2 # Poisson stream rate
531 gamma = 0.5 # Rescheduling rate
532 p = 0.2 # External energy purchase rate
533 e = 0.8 # Battery charging efficiency
534 B_values = [] # To store minimal battery capacities for
different alpha values
535
536 # Initialize the RingGraphSimulator
537 alphas_ring = [0.0, 0.01, 0.05, 0.1]

```

```

537 user1 = Process(alphas=[0.0, 0.01, 0.05, 0.1], gamma=0.5, p
    =0.2, e=0.8, gp=Gillespie(0.5, 0.5))
538 user2 = Process(alphas=[0.0, 0.01, 0.05, 0.1], gamma=0.5, p
    =0.2, e=0.8, gp=Gillespie(0.5, 0.5))
539 user3 = Process(alphas=[0.0, 0.01, 0.05, 0.1], gamma=0.5, p
    =0.2, e=0.8, gp=Gillespie(0.5, 0.5))
540
541 # Initialize the RingGraphSimulator
542 ring_simulator = RingGraphSimulator(num_users=3, alphas=
    alphas_ring, max_lambda=1.0, lambda_step=0.01, ring_rate
    =0.1, gamma=0.5, p=0.2, e=0.8)
543
544 # Add User instances to the simulator
545 ring_simulator.addUser(user1)
546 ring_simulator.addUser(user2)
547 ring_simulator.addUser(user3)
548
549 # Simulate the ring graph scenario
550 ring_simulator.analyze_correlations(gp=ExclusiveGillespie(0.5,
    0.5), amount_of_times=10)
551
552 # Print lambdas
553 ring_simulator.readLambdas()

```

This code defines a simulation framework for studying energy usage and battery capacity within a network of users connected in a ring graph. It comprises several key components, each serving a specific purpose. We tried generating a ring graph of N users each having a background state and a sequence of probabilities depending on the distance. However, this algorithm does not output accurate and uniform results each run.

b)