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):

```python
import numpy as np
import simpy
import matplotlib.pyplot as plt
import copy
import random

import numpy as np

def generate_exponential_times(starttime, endtime, rate,
    num_samples):


    values = np.random.exponential(scale=1/rate, size=
    num_samples)

    waardes = list(values)

    times = []

    current = 0
    for i in range(0, len(waardes)):
        temp = current + waardes[i]
        times.append(temp)
        current = temp


    times = list(times)

    if not all(starttime <= time <= endtime for time in times):
        return generate_exponential_times(starttime, endtime,
    rate, num_samples)
    return times

class Gillespie:

    def __init__(self, p01, p10):
        self.p01 = p01
        self.p10 = p10

```

```
    def calculate_total_rate(self):
        return self.p01 + self.p10

    def Generate(self):
        rando = random.uniform(0, 1)

        if (rando < self.p01/self.calculate_total_rate()):
            return 1
        else:
            return 0

class Singleton:

    def __init__(self):
        self.st = None

    def singleton(self, list):
        if self.st == None:
            self.st = list
            return self.st
        else:
            return self.st

class Process:

    def __init__(self, alphas, gamma, l, p, e, gp = Gillespie
    (0.5,0.5)):
        self.alphas = alphas
        self.gamma = gamma
        self.p = p
        self.e = e
        self.gp = gp
        self.l = l
        self.total_energy_acquired = 1
        self.total_energy_bought = 1
        self.B = 0.1
        self.results = []
        self.amount_of_times = 0
        self.output = None
        self.jobs = []
        self.done = False

    def getOutput(self):

        print("
    --------------------------------------------------")
        print("Simulation Succesfull, running the simulation {
    self.amount_of_times} the values are as follows:")
        for alpha in self.alphas:
            print(f"Alpha: {alpha}, Average Min Battery
    Capacity: {self.results[self.alphas.index(alpha)] / self.
    amount_of_times}")

    def simulate(self, max_capacity, lambda_rate, min_capacity
    = 1000, amount_of_times = 1, distr_value = 200 ,
    capacity_increment = 10, time = 100000):

        results = []
        self.amount_of_times = amount_of_times
        for alpha in self.alphas:
```

```
            results.append(max_capacity)

        # Run the simulations amount_of_times to get a more average result
        for i in range(1, amount_of_times+1):

            arrivals = np.random.poisson(lam=lambda_rate)

            arrival_times = generate_exponential_times(0, time, self.l, arrivals)

            jobs = []
            original_wh = []

            for j in range(0, len(arrival_times)):
                job = []
                tau_k = np.random.exponential(distr_value)
                W_k_new = np.random.exponential(distr_value)
                Wh = W_k_new*tau_k
                original_wh.append(Wh)
                job = [arrival_times[j], tau_k, Wh]
                jobs.append(job)

            self.jobs = tuple(jobs)


            self.output = []
            for i in range(0, len(self.alphas)):
                self.output.append(max_capacity)

            single = Singleton()

            # For every alpha value do the simulation
            for alpha in self.alphas:

                # Find a capacity by iterating over possible values for it
                for capacity in range(max_capacity, min_capacity, -capacity_increment):

                    taak = single.singleton(copy.deepcopy(jobs))

                    self.done = False


                    env = simpy.Environment()

                    # Run the simulation
                    env.process(self.energy_source_process(env, alpha, self.gamma, self.p, self.e, taak, capacity, self.done))

                    # Adjust simulation time accordingly
                    env.run(until=time)

                    if (self.done == True):
                        #results[self.alphas.index(alpha)] += capacity
```

```
                    if (self.output[self.alphas.index(alpha)] > capacity):
                        self.output[self.alphas.index(alpha)] = capacity
                        print(f"using alpha: {alpha}, job {self.alphas.index(alpha)} succesful with current min: {self.output[self.alphas.index(alpha)]}")
                else:
                    print(f"using alpha: {alpha}, job {self.alphas.index(alpha)} failed when attempting capacity: {capacity}, min is {self.output[self.alphas.index(alpha)]}")
                    print("Final", sorted(taak, key=lambda x: x[2], reverse=True))
                    break
        print("--------------------------------------------------")
        print(f"Simulation Succesfull, running the simulation {self.amount_of_times} times, the values are as follows:")
        for alpha in self.alphas:
            print(f"Alpha: {alpha}, Average Min Battery Capacity: {self.output[self.alphas.index(alpha)] / self.amount_of_times}")


    def Execute(self, state, job, capacity, alpha, gamma, p, e):

        res = None
        finished = False
        left = 0

        if state == 1:
            total_source = np.random.uniform(0.01, 2)
            self.total_energy_acquired += total_source
            # Consume energy from the intermittent source
            if job[2] - total_source < 0: # Source is enough to satisfy the job
                remaining_source = total_source - job[2]
                job[2] = 0
                if self.B + remaining_source > capacity: # Excess source cannot be stored
                    self.B = capacity
                    finished = True
                    return res, finished, left
                else:
                    self.B += e * remaining_source
                    finished = True
                    return res, finished, left
            else: # Source is not enough, we need to check the battery
                job[2] = job[2] - total_source
                if self.B - job[2] > 0: # Battery is enough
                    self.B = self.B - job[2]
                    job[2] = 0
                    finished = True
                    res = job[1]
                    return res, finished, left
                else: # Battery is not enough
                    job[2] = job[2] - self.B
                    self.B = 0
```

```
                        left = job[2]
                        res = job[1]
                        return res, finished, left
            # Check if the energy source is ON
        else:
            # Energy source is OFF
            # Determine whether to reschedule or buy external energy
            if self.B - job[2] > 0: # Battery is enough
                    self.B = self.B - job[2]
                    job[2] = 0
                    finished = True
                    return res, finished, left
            else: # Battery is not enough
                job[2] = job[2] - self.B
                self.B = 0
                left = job[2]
                if ((job[2] + self.total_energy_bought) / self.total_energy_acquired <= alpha): #Buy external energy
                    cost = p * job[2]
                    self.total_energy_acquired += job[2]
                    self.total_energy_bought += job[2]
                    job[2] = 0
                    finished = True
                    return res, finished, left
                else:
                    res = gamma
                    return res, finished, left



    def energy_source_process(self, env, alpha, gamma, p, e, tasks, capacity, done):

        queue = []
        self.total_energy_bought = 1
        self.total_energy_acquired = 1

        jobs = copy.deepcopy(tasks)

        while len(jobs) > 0 or len(queue) > 0:


            if (len(queue) > 0):

                job = queue[0]

                yield env.timeout(job[1])

                state = self.gp.Generate()
                logic, finished, left = self.Execute(state, job, capacity, alpha, gamma, p, e)

                if finished:
                    queue.pop(0)
                elif (logic == gamma):
                    rs_time = np.random.exponential(gamma)
                    new_job = [rs_time+job[0], job[1], left]
                    queue.pop(0)
                    if (rs_time+job[0] > env.now):
```

```
                        queue.append(new_job)
                    else:
                        jobs.append(new_job)
                        jobs = sorted(jobs, key=lambda x: x[0])
                elif (logic != None):
                    yield env.timeout(logic)
                else:
                    continue
            else:
                #wait untill time of execution
                if (len(jobs) > 1):
                    for j in range(1, len(jobs)):
                        if (j >= len(jobs)):
                            break
                        if (jobs[j][0] < jobs[0][0]):
                            queue.append(jobs[j])
                            jobs.pop(j)
                        else:
                            break

                yield env.timeout(jobs[0][0])

                if (len(jobs) > 1):
                    if (j >= len(jobs)):
                        break
                    for j in range(1, len(jobs)):
                        if (j >= len(jobs)):
                            break
                        if (jobs[j][0] < jobs[0][0]):
                            queue.append(jobs[j])
                            jobs.pop(j)
                        else:
                            break
                yield env.timeout(jobs[0][1])

                #execute the job logic
                state = self.gp.Generate()
                logic, finished, left = self.Execute(state, jobs[0] , capacity, alpha, gamma, p, e)

                if (finished):
                    jobs.pop(0)
                elif (logic == gamma):
                    rs_time = np.random.exponential(gamma)
                    new_job = [rs_time+jobs[0][0], jobs[0][1], left]
                    jobs.pop(0)
                    jobs.append(new_job)
                    jobs = sorted(jobs, key=lambda x: x[0])
                elif (logic != None):
                    if (len(jobs) > 1):
                        for j in range(1, len(jobs)):
                            if (j >= len(jobs)):
                                break
                            if (jobs[j][0] < logic):
                                queue.append(jobs[j])
                                jobs.pop(j)
                            else:
                                break
                    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, τ and W_k values are generated for each job to be completed within the simulation. τ 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 λ (rate of arrivals), e (Efficiency), and γ (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 α 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 γ 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)

```python
# Code same as part a, above
    def plot_results(self, correlation_values):
        min_battery_capacities = []

        for alpha in self.alphas:
            avg_min_capacity = 0

            for correlation in correlation_values:
                my_time = 0
                job_times = []

                while my_time < 20000:
                    new_job_time = np.random.poisson(lam=lambda_rate) * 100
                    my_time += new_job_time
                    if my_time < 20000:
                        job_times.append(my_time)

                results = []

                for capacity in range(0, 2001, 10):
                    job_times_copy = copy.copy(job_times)
                    env = simpy.Environment()

                    energy_source = env.process(self.energy_source_process(env, alpha, self.gamma, self.p, self.e, job_times_copy, capacity, correlation))

                    env.run(until=20000)

                    if not job_times_copy:
                        results.append(capacity)
                        break
                    else:
                        continue

                if results:  # Only calculate the average when there are results
                    avg_min_capacity += sum(results) / len(results)

            if avg_min_capacity != 0:
                avg_min_capacity /= len(correlation_values)
            min_battery_capacities.append(avg_min_capacity)

        plt.plot(self.alphas, min_battery_capacities, marker='o')
        plt.xlabel('Alpha')
        plt.ylabel('Average Min Battery Capacity')
        plt.title('Influence of Correlation on System Performance')
        plt.grid(True)
```

```
        plt.show()

if __name__ == "__main__":
    lambda_rate = 0.2
    gamma = 0.5
    p = 0.2
    e = 0.8

    p = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
    correlation_values = [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]  # Different correlation levels
    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).

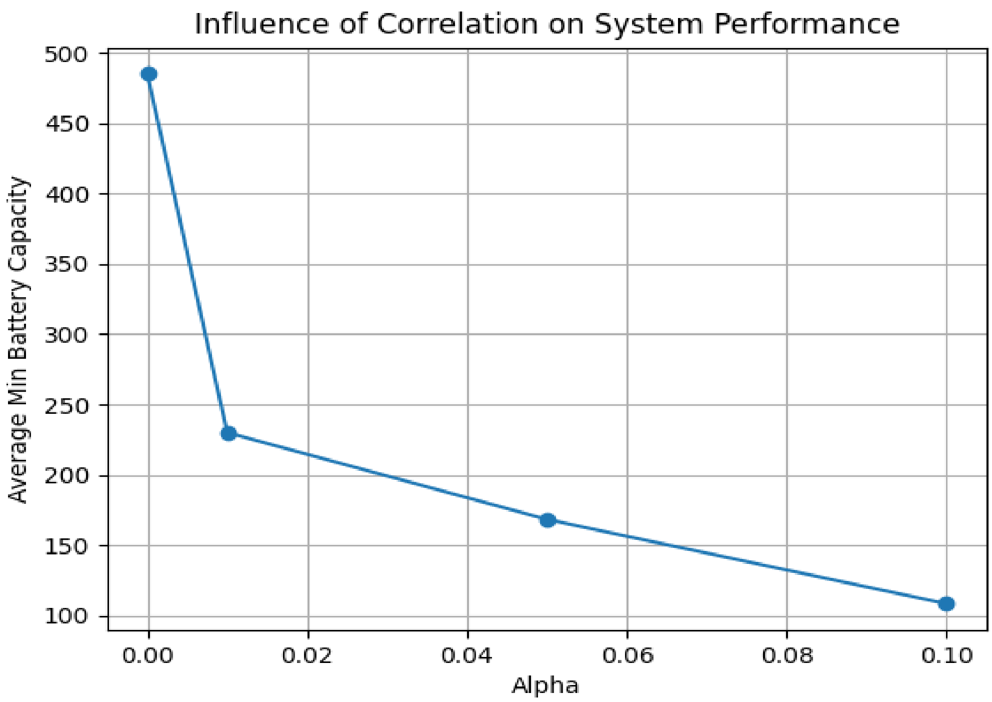

Figure 2: Influence of Correlation on System Performance

2 Scenarios with N users

a)
```python
import numpy as np
import simpy
import matplotlib.pyplot as plt
import copy
import random
import threading

from concurrent.futures import ThreadPoolExecutor, as_completed

class Gillespie(object):

    def __init__(self, p01, p10):
        self.p01 = p01
        self.p10 = p10

    def calculate_total_rate(self):
        return self.p01 + self.p10

    def Generate(self):
        rando = random.uniform(0, 1)
        if (rando < self.p01 / self.calculate_total_rate()):
            return 1
        else:
            return 0

    def setP01(self, p):
        self.p01 = p

    def setP10(self, p):
        self.p10 = p


class ExclusiveGillespie(Gillespie):

    def __init__(self, p01, p10):
        super().__init__(p01, p10)
        self.first = True
        self.lock = threading.Lock()
        self.current = None

    def Generate(self):
        with self.lock:
            if (self.first == True):
                self.first = False
                self.current = super().Generate()
            return self.current

    def next(self):
        with self.lock:
            self.first = True

    def setP01(self, p):
        with self.lock:
            return super().setP01(p)

    def setP10(self, p):
        with self.lock:
            return super().setP10(p)
```

```

class Energy_Collector:

    def __init__(self):
        self.energy = 0
        self.total_additions = 0
        self.total_usage = 0
        self.lock = threading.Lock()

    def getEnergy(self):
        with self.lock:
            return self.energy

    def addEnergy(self, value):
        with self.lock:
            self.energy += value
            self.total_additions += value

    def useEnergy(self, value):
        with self.lock:
            self.energy -= value
            self.total_usage -= value

    def Check(self):
        #debugging
        if (round(self.getEnergy(), 5) == round(self.getTotalAddition() + self.getTotalUsage(), 5)):
            s = 'True, 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))
            print(s)
            return 1
        else:
            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))
            print(s)
            return 0

    def getTotalUsage(self):
        #debugging
        with self.lock:
            return self.total_usage

    def getTotalAddition(self):
        #debugging
        with self.lock:
            return self.total_additions

class Simulator:
    def __init__(self, max_lambda=1.0, lambda_step=0.1):
        self.processes = []
        self.Battery = Energy_Collector()
        self.max_lambda = max_lambda
        self.lambda_step = lambda_step
        self.lambdas = []
```

```
    def addUser(self, process):
        process.setBattery(self.Battery)  # Pass the Battery object to the process
        self.processes.append(process)

    def deleteProcess(self, process):
        self.processes.remove(process)

    def readLambdas(self):
        for i in self.lambdas:
            print(i)

    def simulate(self, max_capacity=2001,  min_capacity = 0, amount_of_times = 1, capacity_increment = 10, time = 20000):
        #Function to simulate different processes per user
        #need to multiproces this function
        #scenario ii

        max_lambda = self.max_lambda
        lambda_step = self.lambda_step
        while max_lambda > 0:
            for user in self.processes:
                user.gp.setP01(max_lambda)
                user.gp.setP10(max_lambda)
            with ThreadPoolExecutor(max_workers=4) as executor:
                futures = [executor.submit(user.simulate, max_capacity, min_capacity, amount_of_times, capacity_increment, time) for user in self.processes]
                for future in futures:
                    future.result()
            backlog = False
            for user in self.processes:
                if user.getResults()[0] > 0:
                    backlog = True
                    break
            if backlog:
                max_lambda -= lambda_step
                break
            else:
                max_lambda += lambda_step
            self.Battery.Check()
        self.lambdas.append(max_lambda)

        for process in self.processes:
            process.getOutput()


    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):
        #scenario i
        for process in self.processes:
            process.setGP(gp)
        self.simulate(max_capacity, min_capacity, amount_of_times, capacity_increment, time)

```

```
    def find_max_lambda(self, max_capacity=2001, min_capacity=0, amount_of_times=1, capacity_increment=10, time=20000):
        max_lambda = self.max_lambda
        lambda_step = self.lambda_step
        while max_lambda > 0:
            for user in self.processes:
                user.gp.setP01(max_lambda)
                user.gp.setP10(max_lambda)
            with ThreadPoolExecutor(max_workers=len(self.processes)) as executor:
                futures = [executor.submit(user.simulate, max_capacity, min_capacity, amount_of_times, capacity_increment, time) for user in self.processes]
                for future in futures:
                    future.result()
            backlog = False
            for user in self.processes:
                if user.getResults()[0] > 0:
                    backlog = True
                    break
            if backlog:
                max_lambda -= lambda_step
                break
            else:
                max_lambda += lambda_step
        return max_lambda

class Process:

    def __init__(self, alphas, gamma, p, e, gp = Gillespie(0.5, 0.5)):
        self.alphas = alphas
        self.gamma = gamma
        self.p = p
        self.e = e
        self.gp = gp
        self.Battery = None
        self.results = None
        self.amount_of_times = 0

    def getResults(self):
        if (self.results == None):
            return "Simulate the process first"
        else:
            return self.results

    def setBattery(self, battery):
        self.Battery = battery

    def setGP(self, gp):
        self.gp = gp

    def energy_source_process(self, env, alpha, gamma, p, e, job_times, capacity):

        state = self.gp.Generate()
        total_energy_demand = 0
        B = 0.1
        total_energy_bought = 0
        jobs = []
```

```
        my_time = 0

        while job_times != []:
            # Generate a new job
            state = self.gp.Generate()

            if job_times and job_times[0] <= my_time:
                tau_k = np.random.exponential(lambda_rate)
                W_k_new = np.random.exponential(2000)  # Random wattage for the job
                Wh_new = tau_k * W_k_new
                job_times.pop(0)
                total_energy_demand += Wh_new
                jobs.append(Wh_new)
            my_time += 10

            if jobs == []:
                continue
            if state == 1:
                total_source = np.random.uniform(0, 10)
                # Consume energy from the intermittent source
                if jobs[0] - total_source < 0: # Source is enough to satisfy the job
                    remaining_source = total_source - jobs[0]
                    jobs[0] = 0
                    jobs.pop(0)
                    if B + remaining_source > capacity: # Excess source cannot be stored
                        self.Battery.addEnergy(remaining_source)
                        B = capacity
                    else:
                        B = B + e * remaining_source
                else: # Source is not enough, we need to check the battery
                    jobs[0] = jobs[0] - total_source
                    if B - jobs[0] > 0: # Battery is enough
                        B = B - jobs[0]
                        jobs[0] = 0
                        jobs.pop(0)
                        yield env.timeout(1 / tau_k)

                    else: # Battery is not enough
                        jobs[0] = jobs[0] - B
                        B = 0
                    yield env.timeout(1 / tau_k)
            # Check if the energy source is ON
            else:
                # Energy source is OFF
                # Determine whether to reschedule or buy external energy
                if B - jobs[0] > 0: # Battery is enough
                        B = B - jobs[0]
                        jobs[0] = 0
                        jobs.pop(0)
                else: # Battery is not enough
                    jobs[0] = jobs[0] - B
                    B = 0
                    if (((jobs[0] + total_energy_bought) / total_energy_demand <= alpha) and (self.Battery.getEnergy()
```

```python
            for alpha in self.alphas:
                print(f"Alpha: {alpha}, Average Min Battery Capacity: {self.results[self.alphas.index(alpha)] / self.amount_of_times}")

lambda_rate = 0.2  # Poisson stream rate
gamma = 0.5  # Rescheduling rate
p = 0.2  # External energy purchase rate
e = 0.8  # Battery charging efficiency
B_values = []  # To store minimal battery capacities for different alpha values

# Create User instances
user1 = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
user2 = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)
user3 = Process([0.0, 0.01, 0.05, 0.1], gamma, p, e)

# Initialize the Simulator
s = Simulator(max_lambda=1.0, lambda_step=0.01)

# Add User instances to the simulator
s.addUser(user1)
s.addUser(user2)
s.addUser(user3)

#s.simulate()
s.simulate_all_same_on_off_process(amount_of_times=10)

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 λ_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 λ 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:

$$E[t_1] = \frac{1}{\lambda} \tag{1}$$

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 identically distributed, so the minimum of them would become:

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

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:

```python
# Import additional libraries
from scipy.spatial.distance import pdist, squareform

import numpy as np
import simpy
import matplotlib.pyplot as plt
import copy
import random
import threading

from concurrent.futures import ThreadPoolExecutor, as_completed

class Gillespie(object):

    def __init__(self, p01, p10):
        self.p01 = p01
        self.p10 = p10

    def calculate_total_rate(self):
        return self.p01 + self.p10

    def Generate(self):
        rando = random.uniform(0, 1)
        if (rando < self.p01 / self.calculate_total_rate()):
            return 1
        else:
            return 0

    def setP01(self, p):
        self.p01 = p

    def setP10(self, p):
        self.p10 = p
```

```


class ExclusiveGillespie(Gillespie):

    def __init__(self, p01, p10):
        super().__init__(p01, p10)
        self.first = True
        self.lock = threading.Lock()
        self.current = None

    def Generate(self):
        with self.lock:
            if (self.first == True):
                self.first = False
                self.current = super().Generate()
            return self.current

    def next(self):
        with self.lock:
            self.first = True

    def setP01(self, p):
        with self.lock:
            return super().setP01(p)

    def setP10(self, p):
        with self.lock:
            return super().setP10(p)

class Energy_Collector:

    def __init__(self):
        self.energy = 0
        self.total_additions = 0
        self.total_usage = 0
        self.lock = threading.Lock()

    def getEnergy(self):
        with self.lock:
            return self.energy

    def addEnergy(self, value):
        with self.lock:
            self.energy += value
            self.total_additions += value

    def useEnergy(self, value):
        with self.lock:
            self.energy -= value
            self.total_usage -= value

    def Check(self):
        #debugging
        if (round(self.getEnergy(), 5) == round(self.
    getTotalAddition() + self.getTotalUsage(), 5)):
            s = 'True, 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))
```

```
            print(s)
            return 1
        else:
            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))
            print(s)
            return 0

    def getTotalUsage(self):
        #debugging
        with self.lock:
            return self.total_usage

    def getTotalAddition(self):
        #debugging
        with self.lock:
            return self.total_additions

class Simulator:
    def __init__(self, max_lambda=1.0, lambda_step=0.1):
        self.processes = []
        self.Battery = Energy_Collector()
        self.max_lambda = max_lambda
        self.lambda_step = lambda_step
        self.lambdas = []

    def addUser(self, process):
        process.setBattery(self.Battery)  # Pass the Battery object to the process
        self.processes.append(process)

    def deleteProcess(self, process):
        self.processes.remove(process)

    def readLambdas(self):
        for i in self.lambdas:
            print(i)

    def simulate(self, max_capacity=2001,  min_capacity = 0, amount_of_times = 1, capacity_increment = 10, time = 20000):
        #Function to simulate different processes per user
        #need to multiproces this function
        #scenario ii

        max_lambda = self.max_lambda
        lambda_step = self.lambda_step
        while max_lambda > 0:
            for user in self.processes:
                user.gp.setP01(max_lambda)
                user.gp.setP10(max_lambda)
            with ThreadPoolExecutor(max_workers=4) as executor:
                futures = [executor.submit(user.simulate, max_capacity, min_capacity, amount_of_times, capacity_increment, time) for user in self.processes]
                for future in futures:
                    future.result()
```

```python
            backlog = False
            for user in self.processes:
                if user.getResults()[0] > 0:
                    backlog = True
                    break
            if backlog:
                max_lambda -= lambda_step
                break
            else:
                max_lambda += lambda_step
            self.Battery.Check()
        self.lambdas.append(max_lambda)

        for process in self.processes:
            process.getOutput()


    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):
        #scenario i
        for process in self.processes:
            process.setGP(gp)
        self.simulate(max_capacity, min_capacity, amount_of_times, capacity_increment, time)

    def find_max_lambda(self, max_capacity=2001, min_capacity=0, amount_of_times=1, capacity_increment=10, time=20000):
        max_lambda = self.max_lambda
        lambda_step = self.lambda_step
        while max_lambda > 0:
            for user in self.processes:
                user.gp.setP01(max_lambda)
                user.gp.setP10(max_lambda)
            with ThreadPoolExecutor(max_workers=len(self.processes)) as executor:
                futures = [executor.submit(user.simulate, max_capacity, min_capacity, amount_of_times, capacity_increment, time) for user in self.processes]
                for future in futures:
                    future.result()
            backlog = False
            for user in self.processes:
                if user.getResults()[0] > 0:
                    backlog = True
                    break
            if backlog:
                max_lambda -= lambda_step
                break
            else:
                max_lambda += lambda_step
        return max_lambda

class Process:

    def __init__(self, alphas, gamma, p, e, gp = Gillespie(0.5, 0.5)):
        self.alphas = alphas
        self.gamma = gamma
```

```python
        self.p = p
        self.e = e
        self.gp = gp
        self.Battery = None
        self.results = None
        self.amount_of_times = 0

    def getResults(self):
        if (self.results == None):
            return "Simulate the process first"
        else:
            return self.results

    def setBattery(self, battery):
        self.Battery = battery

    def setGP(self, gp):
        self.gp = gp

    def energy_source_process(self, env, alpha, gamma, p, e, job_times, capacity):

        state = self.gp.Generate()
        total_energy_demand = 0
        B = 0.1
        total_energy_bought = 0
        jobs = []
        my_time = 0

        while job_times != []:
            # Generate a new job
            state = self.gp.Generate()

            if job_times and job_times[0] <= my_time:
                tau_k = np.random.exponential(lambda_rate)
                W_k_new = np.random.exponential(2000)  # Random wattage for the job
                Wh_new = tau_k * W_k_new
                job_times.pop(0)
                total_energy_demand += Wh_new
                jobs.append(Wh_new)
            my_time += 10

            if jobs == []:
                continue
            if state == 1:
                total_source = np.random.uniform(0, 10)
                # Consume energy from the intermittent source
                if jobs[0] - total_source < 0: # Source is enough to satisfy the job
                    remaining_source = total_source - jobs[0]
                    jobs[0] = 0
                    jobs.pop(0)
                    if B + remaining_source > capacity: # Excess source cannot be stored
                        self.Battery.addEnergy(remaining_source)
                        B = capacity
                    else:
                        B = B + e * remaining_source
```

```
             else: # Source is not enough, we need to check the battery
                 jobs[0] = jobs[0] - total_source
                 if B - jobs[0] > 0: # Battery is enough
                     B = B - jobs[0]
                     jobs[0] = 0
                     jobs.pop(0)
                     yield env.timeout(1 / tau_k)

                 else: # Battery is not enough
                     jobs[0] = jobs[0] - B
                     B = 0
                 yield env.timeout(1 / tau_k)
         # Check if the energy source is ON
         else:
             # Energy source is OFF
             # Determine whether to reschedule or buy external energy
             if B - jobs[0] > 0: # Battery is enough
                     B = B - jobs[0]
                     jobs[0] = 0
                     jobs.pop(0)
             else: # Battery is not enough
                 jobs[0] = jobs[0] - B
                 B = 0
                 if (((jobs[0] + total_energy_bought) / total_energy_demand <= alpha) and (self.Battery.getEnergy() > jobs[0])): #Buy external energy
                     cost = p * jobs[0]
                     total_energy_bought += jobs[0]
                     self.Battery.useEnergy(jobs[0])
                     jobs[0] = 0
                 else:
                     yield env.timeout(np.random.exponential(gamma)) # Reschedule
         #print(test)


 def simulate(self, max_capacity= 2000, min_capacity=0, amount_of_times=1, capacity_increment=10, time=20000):
     results = []
     self.amount_of_times=amount_of_times
     for alpha in self.alphas:
         results.append(0)

     for i in range(1, amount_of_times + 1):
         for alpha in self.alphas:
             my_time = 0
             job_times = []

             while my_time < time:
                 new_job_time = np.random.poisson(lam=lambda_rate) * 100
                 my_time += new_job_time
                 if my_time < time:
                     job_times.append(my_time)

             for capacity in list(range(min_capacity, max_capacity, capacity_increment)):
                 job_times_copy = copy.copy(job_times)
```

```
                    env = simpy.Environment()
                    env.process(self.energy_source_process(env, alpha, self.gamma, self.p, self.e, job_times_copy, capacity))
                    env.run(until=time)  # Adjust simulation time accordingly

                    if not job_times_copy:
                        results[self.alphas.index(alpha)] += capacity
                        break  # Move on to the next alpha
                    else:
                        continue
            if (isinstance(self.gp, ExclusiveGillespie)):
                self.gp.next()

        self.results = results
        #self.getOutput()

    def getOutput(self):

        if (self.Battery.Check()):
            print("--------------------------------------------------")
            print("Simulation Succesfull, running the simulation {self.amount_of_times} the values are as follows:")
            for alpha in self.alphas:
                print(f"Alpha: {alpha}, Average Min Battery Capacity: {self.results[self.alphas.index(alpha)] / self.amount_of_times}")


class RingGraphSimulator(Simulator):
    def __init__(self, num_users, ring_rate, alphas=None, max_lambda=1.0, lambda_step=0.1, **kwargs):
        super().__init__(max_lambda, lambda_step)
        if alphas is None:
            alphas = []  # Set a default value or handle it as needed
        self.alphas = alphas
        self.num_users = num_users
        self.ring_rate = ring_rate
        self.distance_matrix = self.generate_ring_distance_matrix()

        self.processes = []
        self.Battery = Energy_Collector()
        self.gamma = gamma
        self.p = p
        self.e = e
        self.lambdas = []

    def generate_ring_distance_matrix(self):
        positions = np.linspace(0, 2 * np.pi, self.num_users, endpoint=False)
        positions = np.column_stack([np.cos(positions), np.sin(positions)])
        return squareform(pdist(positions))

```

```python
    def generate_ring_correlated_states(self, env, user, alpha, gamma, p, e, job_times, capacity):
        state = user.gp.Generate()
        total_energy_demand = 0
        B = 0.1
        total_energy_bought = 0
        jobs = []
        my_time = 0

        while job_times:
            # Generate a new job
            state = user.gp.Generate()

            if job_times and job_times[0] <= my_time:
                tau_k = np.random.exponential(lambda_rate)
                W_k_new = np.random.exponential(2000)  # Random wattage for the job
                Wh_new = tau_k * W_k_new
                job_times.pop(0)
                total_energy_demand += Wh_new
                jobs.append(Wh_new)
            my_time += 10

            if not jobs:
                continue

            if state == 1:
                total_source = np.random.uniform(0, 10)
                # Consume energy from the intermittent source
                if jobs[0] - total_source < 0:  # Source is enough to satisfy the job
                    remaining_source = total_source - jobs[0]
                    jobs[0] = 0
                    jobs.pop(0)
                    if B + remaining_source > capacity:  # Excess source cannot be stored
                        user.Battery.addEnergy(remaining_source)
                        B = capacity
                    else:
                        B = B + e * remaining_source
                else:  # Source is not enough, we need to check the battery
                    jobs[0] = jobs[0] - total_source
                    if B - jobs[0] > 0:  # Battery is enough
                        B = B - jobs[0]
                        jobs[0] = 0
                        jobs.pop(0)
                        yield env.timeout(1 / tau_k)

                    else:  # Battery is not enough
                        jobs[0] = jobs[0] - B
                        B = 0
                    yield env.timeout(1 / tau_k)
            else:
                # Energy source is OFF
                # Determine whether to reschedule or buy external energy
                if B - jobs[0] > 0:  # Battery is enough
                    B = B - jobs[0]
```

```
                    jobs[0] = 0
                    jobs.pop(0)
                else:  # Battery is not enough
                    jobs[0] = jobs[0] - B
                    B = 0
                    if ((jobs[0] + total_energy_bought) / total_energy_demand <= alpha) and (
                            user.Battery.getEnergy() > jobs[0]):  # Buy external energy
                        cost = p * jobs[0]
                        total_energy_bought += jobs[0]
                        user.Battery.useEnergy(jobs[0])
                        jobs[0] = 0
                    else:
                        yield env.timeout(np.random.exponential(gamma))  # Reschedule

    def simulate_ring_graph(self, gp=ExclusiveGillespie(0.5, 0.5), max_capacity=2001, min_capacity=0,
                            amount_of_times=1, capacity_increment=10, time=20000):
        for process in self.processes:
            process.setGP(gp)

        for user in self.processes:
            env = simpy.Environment()
            user_processes = [self.generate_ring_correlated_states(env, user, user.gp.p01, self.gamma, self.p, self.e,
                                                                  [], max_capacity) for _ in range(amount_of_times)]
            env.process(self.ring_process(env, user_processes, max_capacity, min_capacity, amount_of_times,
                                          capacity_increment, time))
            env.run(until=time)

    def ring_process(self, env, user_processes, max_capacity, min_capacity, amount_of_times, capacity_increment, time):
        for alpha in self.alphas:
            results = []

            for i in range(amount_of_times):
                user_jobs = [list(process) for process in user_processes]

                my_time = 0
                job_times = []

                while my_time < time:
                    my_time += 10
                    yield env.timeout(1)
                    new_job_time = np.random.poisson(lam=lambda_rate) * 100
                    my_time += new_job_time
                    if my_time < time:
                        job_times.append(my_time)

                for capacity in range(min_capacity, max_capacity, capacity_increment):
```

```
                    job_times_copy = copy.copy(job_times)
                    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))]

                    env.run(until=time)  # Adjust simulation time accordingly

                    if not job_times_copy:
                        results.append(sum([self.processes[i].Battery.getEnergy() for i in range(len(self.processes))]))
                        break  # Move on to the next alpha
                    else:
                        continue

            average_result = sum(results) / len(results)
            print(f"Alpha: {alpha}, Average Min Battery Capacity: {average_result}")


    def calculate_temporal_correlation(self, user_processes):
        # Flatten the generator and convert it to a list
        flat_processes = [item for sublist in user_processes for item in sublist]

        # Check if the flattened list is not empty
        if flat_processes:
            # Calculate temporal correlation for a single user
            return np.corrcoef(flat_processes)
        else:
            print("User processes are empty.")
            return np.empty((0, 0))

    def calculate_spatial_correlation(self, user_results):
        # Check if user_results is empty
        if not user_results or not user_results[0]:
            print("Spatial correlation data is empty.")
            return np.empty((0, 0))

        # Calculate spatial correlation between different users
        return np.corrcoef(user_results)

    def plot_temporal_correlation(self, temporal_correlation_matrix):
        # Flatten the matrix to a 1D array
        if temporal_correlation_matrix.size > 0:
            plt.imshow(temporal_correlation_matrix, cmap='viridis', origin='lower', interpolation='none')
            plt.colorbar()
            plt.title('Temporal Correlation Matrix')
            plt.show()
        else:
            print("Temporal correlation matrix is empty.")

    def plot_spatial_correlation(self, spatial_correlation_matrix):
        # Ensure that the matrix is not empty
        if spatial_correlation_matrix.size > 0:
```

```python
            plt.imshow(spatial_correlation_matrix, cmap='viridis', origin='lower', interpolation='none')
            plt.colorbar()
            plt.title('Temporal Correlation Matrix')
            plt.show()
        else:
            print("Temporal correlation matrix is empty.")


    def analyze_correlations(self, gp=ExclusiveGillespie(0.5, 0.5), max_capacity=2001, min_capacity=0,
                             amount_of_times=1, capacity_increment=10, time=20000):
        for process in self.processes:
            process.setGP(gp)

        user_processes = []

        for user in self.processes:
            env = simpy.Environment()
            processes = [self.generate_ring_correlated_states(env, user, user.gp.p01, self.gamma, self.p, self.e,
                                                               [], max_capacity) for _ in range(amount_of_times)]
            env.process(self.ring_process(env, processes, max_capacity, min_capacity, amount_of_times,
                                          capacity_increment, time))
            env.run(until=time)
            user_processes.append(processes)

        # Analyze temporal correlation
        temporal_correlation_matrix = self.calculate_temporal_correlation(user_processes[0][0])
        self.plot_temporal_correlation(temporal_correlation_matrix)

        # Analyze spatial correlation
        user_results = [
        [sum([self.processes[i].Battery.getEnergy() for i in range(len(self.processes))]) for _ in range(amount_of_times)]
        for _ in range(len(self.processes))]
        spatial_correlation_matrix = self.calculate_spatial_correlation(user_results)
        self.plot_spatial_correlation(spatial_correlation_matrix)
# ...

lambda_rate = 0.2  # Poisson stream rate
gamma = 0.5  # Rescheduling rate
p = 0.2  # External energy purchase rate
e = 0.8  # Battery charging efficiency
B_values = []  # To store minimal battery capacities for different alpha values

# Initialize the RingGraphSimulator
alphas_ring = [0.0, 0.01, 0.05, 0.1]

```

```
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))
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))
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))

# Initialize the RingGraphSimulator
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)

# Add User instances to the simulator
ring_simulator.addUser(user1)
ring_simulator.addUser(user2)
ring_simulator.addUser(user3)

# Simulate the ring graph scenario
ring_simulator.analyze_correlations(gp=ExclusiveGillespie(0.5,
    0.5), amount_of_times=10)

# Print lambdas
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)