Queueing theory Assignments

EBB074A05

Nicky D. van Foreest

2022:03:14

CONTENTS

INTRODUCTION

ASSIGNMENTS

These assignments are meant to show you how I build up computer programs (in python) from scratch to analyze and simulate queueing systems. I include the code with the idea that you run it yourself as you read through the tutorial. For that reason many code examples are somewhat long, so that you just have to copy-paste the code to have a fully running example.

I included many (simple) exercises to help you think about the code. As such, most of these exercises ask to explain the code. Of course, for the assignments, you only have to explain the *relevant* parts of the code, that is, part that relate to the queueing model. You should skip, so called, boiler plate code, like importing `numpy`. So keep your answers short; you don't have to win the Nobel prize on literature.

You should read the relevant section of my queueing book to understand what it going on in an assignment. Most of it is very easy, but without background it will be quite cryptic (I believe).

In 2020 I made some youtube movies to illustrate the material. However, this year I revised the code at several places. Hence the movies discuss most (but not all) of the material. You can find the youtube movies here. Below I include the links to each of the videos organized per section.

The document `assignment-answer-template.tex` is the template that you have to use for making your LaTeX assignment.

RUNNING PYTHON

You can install python on your computer (preferred), or run it in a browser if you don't have to install python locally.

- diggy
- colab

There is lots of info on the web on how to setup things.

If you plan to install python on your machine, the best installation is perhaps anaconda.

I prefer to work within emacs (an editor), and run the code in a terminal. This works much faster and more conveniently, but requires a bit (but not much) of intellectual effort and investment in how to configure things on your computer. If you're interested in the power tools I use, check out my tools page.

1

SIMULATION OF QUEUES IN DISCRETE TIME

1.1 INTRO

Here I show how to set up an environment in python to simulate queueing systems in discrete time.

Perhaps you find the following youtube movies helpful.

- https://youtu.be/DfYxayoQmjYc
- https://youtu.be/D8BIAoBICnw
- https://youtu.be/_BoagRyH5c0

1.2 DETERMINISTIC QUEUES

Here we consider a simple queueing system in which all is deterministic.

1.2.1 *one period, demand, service capacity, and queue*

There is one server, jobs enter a queue in front of the server, and the server serves batches of customers, every hour say.

Python Code

```
L = 10
a = 5
d = 8
L = L + a -d
print(L)
```

Python Code

```
L = 3
a = 5
c = 7
d = min(c, L)
L += a -d
print(d, L)
```

1.2.2 *two periods*

Python Code

```
L = 3
a = 5
c = 7
d = min(c, L)
L += a - d

a = 6
d = min(c, L)
L += a - d
print(d, L)
```

Ex 1.2.1. Add a third period, and report your result in the assignment.

1.2.3 *many periods, use vectors and for loops*

Python Code

```
num = 5

a = 9*np.ones(num)
c = 10*np.ones(num)
L = np.zeros_like(a)

L[0] = 20
for i in range(1, num):
    d = min(c[i], L[i-1])
    L[i] = L[i-1] + a[i] - d

print(L)
```

Ex 1.2.2. Run the code for 10 periods and report your result.

Ex 1.2.3. What will be the queue after 100 or so periods?

1.3 STOCHASTIC/RANDOM BEHAVIOR

Real queueing systems show lots of stochasticity: in the number of jobs that arrive in a period, and the time it takes to serve these jobs. In this section we extend the previous models so that we can deal with randomness.

1.3.1 *simulate demand*

Python Code

```
import numpy as np

a = np.random.randint(5, 8, size=5)
print(a)
```

1.3.2 *Set seed*

Python Code

```
import numpy as np

np.random.seed(3)

a = np.random.randint(5, 8, size=5)
print(a)
```

Ex 1.3.1. Why do we set the seed?

1.3.3 *Compute mean and std of simulated queue length for $\rho \approx 1$*

We discuss the concept of load more formally in the course at a later point in time. Conceptually the load is the rate at which work arrives. For instance, if $\lambda = 5$ jobs arrive per hour, and each requires 20 minutes of work (on average), then we say that the load is $5 \times 20/60 = 5/3$. Since one server can do only 1 hour of work per hour, we need at least two servers to deal with this load. We define the utilization ρ as the load divided by the number of servers present.

In discrete time, we define ρ as the average number of jobs arriving per period divided by the average number of jobs we can serve per period. Slightly more formally, for discrete time,

$$\rho \approx \frac{\sum_{k=1}^{n} a_k}{\sum_{k=1}^{n} c_k}. \tag{1}$$

And formally, we should take the limit $n \to \infty$ (but such limits are a bit hard to obtain in simulation).

Python Code

```
import numpy as np

np.random.seed(3)
num = 5000

a = np.random.randint(5, 10, size=num)
c = 7 * np.ones(num)
L = np.zeros_like(a)

L[0] = 20
for i in range(1, num):
    d = min(c[i], L[i-1])
    L[i] = L[i-1] + a[i] - d

print(a.mean(), c.mean(), L.mean(), L.std())
```

Ex 1.3.2. Read the numpy documentation on `numpy.random.randint` to explain the range of random numbers that are given by this function. In view of that, what should `a.mean()` approximately be? Is that larger, equal or smaller than `c`?

1.3.4 *plot the queue length process*

Glue the next code after the other code.

Python Code

```
import matplotlib.pyplot as plt

plt.clf()
plt.plot(L)
plt.savefig('figures/queue-discrete_1.pdf')
```

Ex 1.3.3. Comment on the graph you get. Did you expect such large excursions of the queue length process?

1.3.5 *A trap: integers versus floats*

Suppose that we change the arrival rate a bit.

Python Code

```
num = 5000

np.random.seed(3)
a = np.random.randint(5, 9, size=num)
c = (5+8)/2 * np.ones(num)
L = np.zeros_like(a)

L[0] = 20
for i in range(1, num):
    d = min(c[i], L[i-1])
    L[i] = L[i-1] + a[i] - d


plt.clf()
plt.plot(L)
plt.savefig('figures/queue-discrete-1-1.pdf')
```

Don't forget that the `numpy` library has to be imported (as we did before).

Ex 1.3.4. Compare the definition of `a` and `c` to what we had earlier. What is `a.mean()` now approximately? Observe that the mean of `c` is now around 6.5.

When you make the plot you should see that is is very different from the one before. To explain why this is so, the following somewhat cryptic question will be helpful, hopefully.

Ex 1.3.5. What is $9 - 6.5$? What is `int(9-6.5)`, that is, run the code in python, and type in the answer in your answer sheet. Explain the difference between these two numbers? Explain that since `L` stores *integers*, not floats, we actually use a service capacity of 7, *not* of 6.5.

Ex 1.3.6. The code below repairs the above problem. Explain which line is different from the previous code. How did that change repair the problem? Now explain the title of this section. (BTW, I made this type of error with floats and ints many, many times. The reason to include these exercises is to make you aware of the problem, so that you can spot it when you make the same error.)

Include the graph in your report and explain the differences.

Python Code

```
num = 5000

np.random.seed(3)
a = np.random.randint(5, 9, size=num)
c = (5+8)/2 * np.ones(num)
L = np.zeros_like(a, dtype=float)

L[0] = 20
for i in range(1, num):
    d = min(c[i], L[i-1])
    L[i] = L[i-1] + a[i] - d


plt.clf()
plt.plot(L)
plt.savefig('figures/queue-discrete-1-2.pdf')
```

1.3.6 *Serve arrivals in the same period as they arrive*

Ex 1.3.7. Change the code such that the arrivals that occur in period i can also be served in period i. Include your code in your assignment. Then make a graph (include that too of course) and compare your results with the results of the simulation we do here (recall, in the simulations up to now, arrivals cannot be served in the periods in which they arrive).

1.3.7 *Drift when $\rho > 1$*

Python Code

```
num = 5_000

np.random.seed(3)
a = np.random.randint(5, 9, size=num)
c = (5+8)/2.3 * np.ones(num)
L = np.zeros_like(a, dtype=float)

L[0] = 20
for i in range(1, num):
    d = min(c[i], L[i-1])
    L[i] = L[i-1] + a[i] - d


plt.clf()
plt.plot(L)
plt.savefig('figures/queue-discrete_2.pdf')
```

Ex 1.3.8. Include the graph in your report. What is `c.mean()-a.mean()`? Explain the slope of the line (fitted through the points.) (Check the hint.)

1.3.8 *Drift when $\rho < 1$, start with large queue.*

Python Code

```
num = 5_000

np.random.seed(3)
a = np.random.randint(5, 9, size=num)
c = (5+8)/1.8 * np.ones(num)
L = np.zeros_like(a, dtype=float)

L[0] = 2_000
for i in range(1, num):
    d = min(c[i], L[i-1])
    L[i] = L[i-1] + a[i] - d


plt.clf()
plt.plot(L)
plt.savefig('figures/queue-discrete_3.pdf')
```

Ex 1.3.9. Explain the slope of the lines (if you would fit that through the points.)

1.3.9 *Hitting times*

When ρ 1 and L_0 is some large number we could be interested in estimating the time until the queue is empty. With this we can decide if extra capacity hired to remove the long queue (for instance, long waiting times in a hospital) suffices. If the time to hit zero is still too long, we should hire more capacity.

Here we study how to estimate the hitting time $\tau = \min\{k : Z_k \leq 0\}$, where $Z_k = Z_{k-1} + a_k - c_k$. (Observe that Z and L are not the same everywhere: Z can become negative, the number L in the system L is always ≥ 0.)

Ex 1.3.10. Run the following code and include the figure in your report. Use the CTL (see the hint if you forgot) to explain that most sample paths of Z seem to hit 0 when

$$\tau \approx \frac{100}{(5+9)/1.9 - 7}.$$

Next, change the factor 1.9 to 1.2. Why do the graphs lie much nearer to each other?

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

num = 500
L0 = 100


def hitting_time():
    a = np.random.randint(5, 10, size=num)
    c = (5 + 9) / 1.9 * np.ones(num)
    a[0] = L0
    Z = (a - c).cumsum()
    plt.plot(Z)
    return


samples = 30
for n in range(samples):
    hitting_time()

plt.savefig("figures/free-random-walk.pdf")
```

We see in the figure of the previous exercise that there is considerable variation in the time the random walk hits zero—a bit more specifically, hits the set $\{\ldots, -2, -1, 0\}$—when ρ is not very small. We need some code to compute τ for a sample path of Z.

Ex 1.3.11. Explain how the loop over `i` in the function `hitting_time` computes τ for a specific sample path. You don't have to run the code, the aim is that you understand how the algorithm works.

Python Code

```
import numpy as np

np.random.seed(3)

num = 500
L0 = 100
```

```

def hitting_time():
    a = np.random.randint(5, 10, size=num)
    c = (5 + 9) / 1.9 * np.ones(num)
    L = L0
    for i in range(1, num):
        L += +a[i] - c[i]
        if L <= 0:
            return i


samples = 10
tau = np.zeros(samples)
for n in range(samples):
    tau[n] = hitting_time()

print(tau.mean(), tau.std())
```

Once again, python (and R) are rather slow in comparison to C or fortran; the factor in speed can be 100 or more in the execution of for loops. For this reason I prefer to tinker a bit with code to push as much of the computation to numpy.

Ex 1.3.12. Run this code, and explain the output of each print statement of this piece of code. In particular, realize that for `np.argmax`: 'In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.' (I found this explanation in the numpy documentation.)

Python Code

```
import numpy as np

np.random.seed(3)

num = 10
L0 = 10
samples = 3


dims = (samples, num)
a = np.random.randint(5, 10, size=dims)
c = 8 * np.ones(dims)
a[:, 0] = L0
Z = (a - c).cumsum(axis=1)
print(Z)
print(Z <= 0)
print(np.argmax(Z <= 0, axis=1))
```

BTW, observe that I use small samples to print the output, so that it's easy to see how everything works.

Here is the final code.

Python Code

```
import numpy as np
from scipy.stats import norm
```

```python
import matplotlib.pyplot as plt

np.random.seed(3)

num = 400
L0 = 100
samples = 3000


dims = (samples, num)
a = np.random.randint(5, 10, size=dims)
c = (5 + 9) / 1.9 * np.ones(dims)
a[:, 0] = L0
Z = (a - c).cumsum(axis=1)
tau = np.argmax(Z <= 0, axis=1)
print(tau.mean(), tau.std())

tau_scaled = (tau - tau.mean()) / tau.std()
print(tau_scaled.mean(), tau_scaled.std())
bins = np.linspace(-3, 3, 40)

B = (bins[1:] + bins[:-1]) / 2

plt.hist(tau_scaled, bins=bins, density=True)
plt.plot(B, norm.pdf(B))
plt.savefig("figures/tau.pdf")
```

Ex 1.3.13. Use the CLT to provide some intuition to see why `tau_scaled` is approximately normally distributed. Include the plot in your report.

Here are some specific points of interest in the code:

1. `num=400` just to guess to ensure that `Z[400] < 0` for all sample paths. This trick allowed me to use numpy. Otherwise I would have needed a for loop in python, which I wanted to avoid.
2. `bins` contains the edges of the bins. To plot the pdf of the standard normal distribution, I need the mid points of the bins. This explains `B`.

1.3.10 *Things to memorize*

1. If the capacity is equal or less than the arrival rate, the queue length will explode.
2. If the capacity is larger than the arrival rate, the queue length will 'stay around 0', roughly speaking.
3. If we start with a huge queue, but the service capacity is larger than the arrival rate, then the queue will drain like a straight line (roughly).

1.4 QUEUES WITH BLOCKING

Consider a queue that is subject to blocking: this means that when the queue exceeds K, say, then the excess jobs are rejected.

1.4.1 *A simple rejection rule*

Python Code

```
num = 500

np.random.seed(3)
a = np.random.randint(0, 20, size=num)
c = 10*np.ones(num)
L = np.zeros_like(a)
loss = np.zeros_like(a)

K = 30 # max people in queue, otherwise they leave

L[0] = 28
for i in range(1, num):
    d = min(c[i], L[i-1])
    loss[i] = max(L[i-1] + a[i] - d - K, 0)  # this
    L[i] = L[i-1] + a[i] - d - loss[i] # this 2

lost_fraction = sum(loss)/sum(a)
print(lost_fraction)
```

Ex 1.4.1. Explain how the line marked as `this` works, in other words, how does that line implement a queue with loss? In line `this 2` we subtract `loss[i]`; why?

Ex 1.4.2. Why is, in this case, `dtype=float` not necessary in the definition of `L`?

Ex 1.4.3. Add the following code to make the graph of the (simulated) queue length process.

Python Code

```
plt.clf()
plt.plot(L)
plt.savefig('figures/queue-discrete-loss.pdf')
```

Include the graph in your report. Does the blocking rule work as it should?

1.4.2 *Rejection at the start of the period*

If we would assume that rejection occurs as the start of the period, the code has to be as follows:

Python Code

```
for i in range(1, num):
    d = min(c[i], L[i-1])
    loss[i] = max(L[i-1] + a[i] - K, 0)
    L[i] = L[i-1] + a[i] - d - loss[i]

lost_fraction = sum(loss)/sum(a)
print(lost_fraction)
```

Ex 1.4.4. Explain the line in the code that changed.

Ex 1.4.5. Explain that this rule has the same effect as assuming that departures occur after the rejection.

1.4.3 *Estimating rejection probabilities*

With the code below we can estimate the distribution $p_i = \mathsf{P}[L = i]$.

Python Code

```python
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(3)

num = 5000

np.random.seed(3)
a = np.random.randint(0, 18, size=num)
c = 10 * np.ones(num)
L = np.zeros_like(a)

K = 30
p = np.zeros(K + 1)

L[0] = 28
for i in range(1, num):
    d = min(c[i], L[i - 1])
    L[i] = min(L[i - 1] + a[i] - d, K)
    p[L[i]] += 1

plt.clf()
plt.plot(p)
plt.savefig('figures/queue-discrete-loss-p.pdf')
```

Ex 1.4.6. Why should `p` have a length of `K+1`? Then explain how the code estimates p_i.

Ex 1.4.7. Note that `p` is not normalized (i.e., sums up to 1). The following code repairs that. Explain how it works.

Python Code

```python
p /= p.sum()
```

1.4.4 *Rejection probabilities under high loads*

Ex 1.4.8. Change the parameters such $\rho = 1.51$. Then make again a plot of p. (Just copy my code, but change the parameters or the distribution of `a`, or `c`.)

Ex 1.4.9. Now change the parameters such $\rho \approx 10$, use a simple argument to show that π_{K-2} should be really small. Check this intuition by doing a simulation with the appropriate parameters. Include your code and the graph of `p`.

2

QUEUEING CONTROL: PSYCHIATRISTS DOING INTAKES

2.1 INTRO

There are 5 psychiatrists doing intakes. In their current organization, the queue of patients waiting for intakes is way too long, much longer then they like to see. Here I consider some strategies to deal with this controlling the queue length process, and I use simulation to evaluate how successful these are.

- https://youtu.be/bCU3oP6r-00

2.2 BASE SITUATION

Five psychiatrists do intakes. See the queueing book for further background.

2.2.1 *Load standard modules*

We need some standard libraries for numerical work and plotting.

Python Code

```
import numpy as np
import matplotlib.pylab as plt
from matplotlib import style

style.use('ggplot')

np.random.seed(3)
```

2.2.2 *Simulate queue length*

Python Code

```
def computeQ(a, c, Q0=0):  # initial queue length is 0
    N = len(a)
    Q = np.empty(N)  # make a list to store the values of  Q
    Q[0] = Q0
    for n in range(1, N):
        d = min(Q[n - 1], c[n])
        Q[n] = Q[n - 1] + a[n] - d
    return Q
```

Ex 2.2.1. Compute the queue lenghts when

Python Code

```
a = [1,2,5,6,8,3,7,3]
c = [2,2,0,5,4,4,3,2]
```

and include the results in your report.

2.2.3 *Arrivals*

We start with run length 10 for demo purpose; later we extend to longer run times

Python Code

```
a = np.random.poisson(11.8, 12)
print(a)
```

2.2.4 *Service capacity*

Python Code

```
def unbalanced(a):
    p = np.empty([5, len(a)])
    p[0, :] = 1.0 * np.ones_like(a)
    p[1, :] = 1.0 * np.ones_like(a)
    p[2, :] = 1.0 * np.ones_like(a)
    p[3, :] = 3.0 * np.ones_like(a)
    p[4, :] = 9.0 * np.ones_like(a)
    return p

p = unbalanced(a)
print(p)
```

Ex 2.2.2. Which psychiatrist does the most intakes per week?

2.2.5 *Include holidays*

Python Code

```
def spread_holidays(p):
    for j in range(len(a)):
        psych = j % 5
        p[psych, j] = 0

spread_holidays(p)
print(p)
```

Ex 2.2.3. What is the long-run time average of the weekly capacity?

2.2.6 *Total weekly service capacity*

Python Code

```
s = np.sum(p, axis=0)
print(s)
```

Ex 2.2.4. Explain why we need to take the sum over `axis=0` to compute the average weekly capacity for the intakes.

2.2.7 *Simulate the queue length process*

Python Code

```
np.random.seed(3)

```

```
a = np.random.poisson(11.8, 1000)
p = unbalanced(a)
spread_holidays(p)
s = np.sum(p, axis=0)

Q1 = computeQ(a, s)

plt.clf()
plt.plot(Q1)
plt.savefig("figures/psych1.pdf")
```

Ex 2.2.5. Choose your own seed, run the code, include the figure in your report and comment on what you see.

2.3 EVALUATION OF BETTER (?) PLANS

2.3.1 *Balance the capacity more evenly over the psychiatrists*

I set the seed to enforce a start with the same arrival pattern.

#+begin$_{\text{src}}$ python def balanced(a): p = np.empty([5, len(a)]) p[0, :] = 2.0 * np.ones$_{\text{like}}$(a) p[1, :] = 2.0 * np.ones$_{\text{like}}$(a) p[2, :] = 3.0 * np.ones$_{\text{like}}$(a) p[3, :] = 4.0 * np.ones$_{\text{like}}$(a) p[4, :] = 4.0 * np.ones$_{\text{like}}$(a) return p

np.random.seed(3) a = np.random.poisson(11.8, 1000)

p = balanced(a) spread$_{\text{holidays}}$(p) s = np.sum(p, axis=0) Q2 = computeQ(a, s)

plt.plot(Q2) plt.savefig("figures/psych2.pdf") #+end$_{\text{exercise}}$

Ex 2.3.1. How can we see that the effect of balancing capacity is totally uninteresting?

Ex 2.3.2. Change the capacities of the psychiatrists but keep the average weekly capacity the same. Include a graph of your result, and comment on the effect of your changes.

2.3.2 *Synchronize holidays*

Supposwe we would synchronize the holidays so that all psychiatrists take holiday in the same week. Would that have an effect on the queue process?

Python Code

```
a = np.random.poisson(11.8, 12)


def synchronize_holidays(p):
    for j in range(len(a) // 5 + 1):
        p[:, 5 * j] = 0  # this
    return p

p = unbalanced(a)
p = synchronize_holidays(p)
print(p)
```

Ex 2.3.3. Explain how the code works. Specifically, what does the line marked as `this`?

Let's do a longer run to see the effect.

Ex 2.3.4. In the code below, choose your own seed, run it, include the figure in your report and comment on what you see.

Python Code

```
np.random.seed(3)

a = np.random.poisson(11.8, 1000)
p = unbalanced(a)
spread_holidays(p)
s = np.sum(p, axis=0)
Q3 = computeQ(a, s)

plt.clf()
plt.plot(Q3)

p = balanced(a)
synchronize_holidays(p)
s = np.sum(p, axis=0)
Q4 = computeQ(a, s)

plt.plot(Q4)
plt.savefig("figures/psych3.pdf")
```

Ex 2.3.5. Change the code such that psychiatrists go on holiday every 6 weeks. However, modify the weekly capacities of the psychiatrists such that the total average weekly capacity remains the same. Include your code, and check with a sum (over an appropriate axis) that the average weekly capacity is still the same after your changes.

Ex 2.3.6. Just to improve your coding skills (and your creativity), formulate another vacation plan. Implement this idea in code, and test its success/failure. Make a graph to show its effect on the dynamics of the queue length. (I don't mind whether your proposal works or not; as long as you 'play' and investigate, all goes.) Include your code—if you ported all this code to R , then include your R code— and comment on the difficult points.

Most probably, your proposals will also not solve the problem. We need something smarter.

2.4 CONTROL CAPACITY AS A FUNCTION OF QUEUE LENGTH

2.4.1 *Simple on-off strategies*

Let's steal an idea from supermarkets: dynamic control.

Python Code

```
lower_thres = 12
upper_thres = 24

def computeQExtra(a, c, e, Q0=0):  #  initial queue length is 0
    N = len(a)
    Q = [0] * N  # make a list to store the values of  Q
    Q[0] = Q0
    for n in range(1, N):
        if Q[n - 1] < lower_thres:
            C = c - e
```

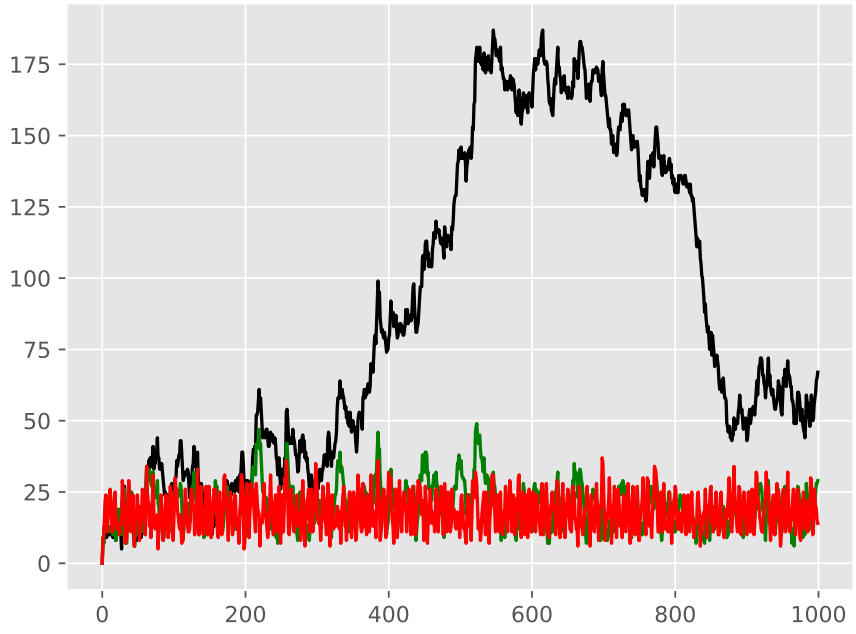
```
        elif Q[n-1] >= upper_thres:
            C = c + e
        d = min(Q[n-1], C)
        Q[n] = Q[n-1] + a[n] -d
    return Q


np.random.seed(3)
a = np.random.poisson(11.8, 1000)
c = 12
Q = computeQ(a, c * np.ones_like(a))
Qe1 = computeQExtra(a, c, 1)
Qe5 = computeQExtra(a, c, 5)

plt.clf()
plt.plot(Q, label="Q", color='black')
plt.plot(Qe1, label="Qe1", color='green')
plt.plot(Qe5, label="Qe5", color='red')
plt.savefig("figures/psychfinal.pdf")
```

Ex 2.4.1. Explain how the if statements in the code above work.

Ex 2.4.2. Explain how this idea relates to what happens in a supermarket if there are still open service desks but queues become very long.



We see, dynamically controlling the service capacity (as a function of queue length) is a much better plan.

Ex 2.4.3. Use simulation to show that the psychiatrists don't have more work.

Ex 2.4.4. Choose some other control thresholds (something reasonable of course, but otherwise you are free to select your own values.) Run the simulation with your values, include a graph and explain what you see.

2.4.2 *Hire an extra server for a fixed amount of time*

In the real case the psychiatrists hired an extra person to do intakes when the queue became very long, 100 or higher, and then they hired this person for one month (you may assume that a month consists of 4 weeks). Suppose this person can do 2 intakes a day and works for 4 days a week.

The code below implements this control algorithm.

Ex 2.4.5. Explain the code below.

Python Code

```
import numpy as np
import matplotlib.pylab as plt
from matplotlib import style

style.use('ggplot')
np.random.seed(3)

extra_capacity = 8  # extra weekly capacity
contract_duration = 4  # weeks


def compute_Q_control(a, c, Q0=0):
    N = len(a)
    Q = np.empty(N)
    Q[0] = Q0
    extra = False
    mark_time = 0
    for n in range(1, N):
        if Q[n - 1] > 100:
            extra = True
            mark_time = n
        if extra and n >= mark_time + contract_duration:
            extra = False
        d = min(Q[n - 1], c[n] + extra * extra_capacity)
        Q[n] = Q[n - 1] + a[n] - d
    return Q


a = np.random.poisson(11.8, 1000)
c = 12
Q = compute_Q_control(a, c * np.ones_like(a), Q0=110)
# print(Q)
plt.clf()
plt.plot(Q, label="Q", color='black')
plt.savefig("figures/psych_extra.pdf")
```

Ex 2.4.6. Do a number of experiments to see the effect of the duration of the contract by making it longer (experiment 1), or shorter (experiment 2). Run the simulation, Include graphs, and discuss the effect of these changes.

Ex 2.4.7. Now change the number of intakes per day done by the extra person. (For instance, an experienced person can do more intakes in the same amount of time than a newbie. However, this comes at an additional cost of course.) Make a graph, and compare the effect of this change to the previous (changing the duration).

Ex 2.4.8. If you were a consultant, what would you advice the psychiatrists on how to control their waiting lists?

3 SIMULATION OF QUEUEING PROCESSES IN CONTINUOUS TIME

3.1 INTRO

I discuss two elegant algorithms to simulate the waiting time process. One is for a system with one server, and jobs are served in the order in which they arrive. The second is for a multi-server FIFO queue.

- YouTube: https://youtu.be/h1OTvdLs9ik

3.2 COMPUTING WAITING TIMES

Here we just follow the steps of the queueing book to construct a single server FIFO queue in continuous time and compute the waiting and sojourn times.

3.2.1 *Load standard modules*

We need the standard libraries for numerical work and plotting.

Python Code

```
import numpy as np
import matplotlib.pylab as plt
from matplotlib import style

np.random.seed(3)
```

3.2.2 *Inter-arrival times*

Simulate random interarrival times that are $\sim \text{Exp}(\lambda)$, with $\lambda = 3$. First I take just three jobs, so that I can print out all intermediate results and check how things work. Once I am convinced about the correctness, I run a simulation for many jobs.

Python Code

```
num = 3
labda = 3
X = np.random.exponential(scale=labda, size=num)
print(X)
```

Here is an important check (I always forget the meaning of λ when I provide it to the simulator)

Python Code

```
num = 100
labda = 3
X = np.random.exponential(scale=labda, size=num)
print(X.mean())
```

Ex 3.2.1. Explain that `scale=labda` sets the interarrival times to 3, but that in our queueing models, λ should correspond to the arrival rate. Why is the code below in line with what we want?

Python Code

```
num = 3
labda = 3
X = np.random.exponential(scale=1/labda, size=num)
```

3.2.3 *Arrival times*

Ex 3.2.2. Why do we generate first random inter-arrival times, and use these to compute the arrival times? Why not directly generate random arrival times?

Python Code

```
A = X.cumsum()
print(A)
```

Check the output to see that the arrival time of job 0 is A_0 0. But I want time to start at time $A_0 = 0$. Here is the trick to achieve that.

Python Code

```
X[0] = 0
A = X.cumsum()
print(A)
```

Ex 3.2.3.
1. Why is this better?
2. Why can we remove `X[0]` without fundementally changing the analysis?

3.2.4 *Service times*

We have arrival times. We next need the service times of the jobs. Assume they are $\sim \text{Exp}(\mu)$ with μ somewhat larger than λ. (Recall this means that jobs can be served faster than that they arrive.)

Python Code

```
mu = 1.2 * labda
S = np.random.exponential(scale=1/mu,size=len(A))
S[0] = 0
print(S)
```

Note, `S[0]` remains unused; it should correspond to job 0, but we neglect this job 0 in the remainder.

Ex 3.2.4. Why do I use `size=len(A)` in the definition of `S`?

Ex 3.2.5. Why do we set `scale=1/mu`?

Ex 3.2.6. It's easy to compute the mean service time like this

Python Code
```
print(S.mean())
```

Explain that this is not exactly equal to $\mathsf{E}[S]$.

3.2.5 *Departure times*

The standard recursion to compute the departure times.

Python Code
```
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]

print(D)
```

3.2.6 *Sojourn times*

How long do you stay in the system if you arrive at some time A_n and you depart at D_n?

Python Code
```
J = D - A
print(J)
```

3.2.7 *Waiting times*

If your sojourn time is 10, say, and your service time at the server is 3 (and there is just one server and the service discipline is FIFO), then what was your time in queue?

Python Code
```
W = J - S
print(W)
```

Ex 3.2.7. Recall that we set `S[0] = 0`. Suppose that we wouldn't have done this, and we would run the simulation for a small number of cases, why can `W.mean()` be negative?

3.3 KPIS AND PLOTTING

The next step is to see how to compute the most important indicators to assess the performance of the system. We can use these, so-called, Key Performance Indicators (KPIs) to see whether we should add, for instance, service capacity.

3.3.1 *Relevant averages*

Python Code
```
print(W.mean(), W.std())
```

Python Code
```
print(J.mean(), J.std())
```

Ex 3.3.1. The mean of W is not entirely correct in the way we compute it here. What is (just a bit) wrong?

Python Code

```
plt.clf()
plt.plot(J)
plt.savefig("figures/sojourn.pdf")
```

Ex 3.3.2. Change the simulation length to 1000 jobs. Do one run for $\mu = 3.5$ and another for 2.8. Compute the KPIs, make a plot, and include that in your assignment. Comment on what you see.

3.3.2 *Server KPI: idle time*

This code computes the total time the server is idle, and then computes the fraction of time the server is idle.

Python Code

```
rho = S.sum() / D[-1]
idle = (D[-1] - S.sum()) / D[-1]
print(idle)
```

Ex 3.3.3. Explain the code above. Some specific points:

1. Why is `S.sum()` the total busy time of the server?
2. Why do we divide by `D[-1]` in the computation of ρ?
3. Explain the computation of the `idle` variable.

The next code computes the separate idle times.

Python Code

```
idle_times = np.maximum(A[1:] - D[:-1], 0)
print(idle_times)
print(idle_times.sum())
print(D[-1] - S.sum())
```

Ex 3.3.4. Run this code for a simulation with 10 or so jobs (some other small number). Explain how this code works. Which line is a check on the computations?

3.3.3 *Server KPI: busy time*

We also like to know how a long the server has to work uninterruptedly. Finding the busy times is quite a bit harder than the idle times. (A busy time starts when a job arrives at an empty system and it stops when the server becomes free again.)

Ex 3.3.5. To help you understand the code, let's first do a numerical example. Suppose jobs 1,4,8 find an empty system upon *arrival*. The simulation contains 10 jobs. Why do jobs 3,7,10 leave an empty system behind upon *departure*?

With this idea, we can compute the idle times in another way (as a check on earlier work), and then we extend the approach to the busy times.

Python Code

```
import numpy as np

np.set_printoptions(suppress=True)
np.random.seed(3)

num = 10
labda = 3
X = np.random.exponential(scale=1 / labda, size=num)
X[0] = 0
A = X.cumsum()
mu = 1.2 * labda
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]


W = D - S - A # waiting times
idx = np.argwhere(np.isclose(W, 0))
idx = idx[1:]  # strip A[0]
idle_times = np.maximum(A[idx] - D[idx - 1], 0)
print(idle_times.sum())
```

Ex 3.3.6. What is stored in `idx`? Why do we strip `A[0]`? Why do we subtract `D[idx-1]` and not `D[idx]`? (Print out the variables to understand what they mean, e.g., `print(idx)`.)

Now put the next piece of code behind the previous code so that we can compute the busy times.

Python Code

```
busy_times = D[idx - 1][1:] - A[idx][:-1]
last_busy = D[-1] - A[idx[-1]]
print(busy_times.sum() + last_busy, S.sum())
```

Ex 3.3.7. Explain these lines. About the last line, explain why this acts as a check.

3.3.4 *Virtual waiting time*

Plotting the virtual waiting time is subtle. (The code below is short, hence may seem to be easy to find, but for me it wasn't. To get it right took me two to three hours, and I also discovered other bugs I made elsewhere. Coding is hard!)

Ex 3.3.8. Make a plot of the virtual waiting time by hand (you don't have a make a large plot, just show that you understand what the virtual waiting process looks like). Find out which points are the most important ones to characterize the virtual waiting times, and explain why this is so.

Here is the code to plot the virtual waiting time.

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

```

```
np.random.seed(3)

num = 40
labda = 1
mu = 1.1 * labda
T = 10  # this acts as the threshold
X = np.random.exponential(scale=1 / labda, size=num)
X[0] = 0
A = np.zeros_like(X)
A = X.cumsum()
S = np.ones(len(A)) / mu
S[0] = 0
D = np.zeros_like(A)

W = np.zeros_like(A)
for k in range(1, len(X)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)
    D[k] = A[k] + W[k] + S[k]

idx = np.where(W <= 0)[0] # this
empty = D[idx[1:] - 1]

E = np.zeros((2 * len(A) + len(empty), 2))  # epochs
E[: len(A), 0] = A
E[: len(A), 1] = W
E[len(A) : 2 * len(A), 0] = A
E[len(A) : 2 * len(A), 1] = W + S
E[2 * len(A) : 2 * len(A) + len(empty), 0] = empty
E[2 * len(A) : 2 * len(A) + len(empty), 1] = 0
E = E[np.lexsort((E[:, 1], E[:, 0]))]

plt.plot(E[:, 0], E[:, 1])
plt.savefig("figures/virtual-waiting-time.pdf")
```

The `this` line is perhaps somewhat strange. By printing the result, we can find out that `np.where(W <= 0)` returns a tuple of which the first element is an array of the indices where `W` is zero. To get that first element we add the extra `[0]`.

Ex 3.3.9. 1. Explain how we store the relevant epochs in `E`.

2. Why do we use `idx[1:]` (What is `W[0]`)?
3. Why do we subtract 1 from `idx[1:]`?
4. Why do we use `np.lexsort`? (Check the documentation to see how lexical sorting works. It is important to know what lexical sorting is.)

3.4 COMPUTING THE NUMBER OF JOBS IN THE SYSTEM

We have the waiting times, but not the number of jobs in the system (queue). Here we show how to plot L, i.e., the number of jobs in the system as seen by a job *upon* arrival.

A simple, but *inefficient*, algorithm to construct L is the following.

Python Code

```
import numpy as np

```

```
np.random.seed(3)

num = 10
labda = 1
mu = 1.1 * labda

X = np.random.exponential(scale=1 / labda, size=num)
X[0] = 0
A = np.zeros_like(X)
A = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]

L = np.zeros_like(A)
for k in range(1, len(A)):
    idx = 0
    while D[idx] < A[k]:
        idx += 1
    L[k] = k - idx

print(L)
```

Ex 3.4.1. 1. Explain how this algorithm works.

2. Why does this algorithm find the number of jobs as seen *just before the arrival* of a job?
3. What line should be changed so that we count the number of jobs in the system *just after* the arrival? What should it become?
4. Why is this algorithm (very) inefficient?

Making the algorithm much more efficient is not hard.

Python Code

```
L = np.zeros_like(A)
idx = 0
for k in range(1, len(A)):
    while D[idx] < A[k]:
        idx += 1
    L[k] = k - idx
```

Ex 3.4.2. Why is this much better?

Ex 3.4.3. Here is another algorithm that plots $\{L(t)\}$. Explain how it works, and its difference with the previous algorithm.

Python Code

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(3)

num = 4
```

```
labda = 3
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.3 * labda
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]

L = np.zeros((len(A) + len(D), 2))
L[: len(A), 0] = A
L[1 : len(A), 1] = 1
L[len(D) :, 0] = D
L[len(D) + 1 :, 1] = -1
N = np.argsort(L[:, 0], axis=0)
L = L[N]
L[:, 1] = L[:, 1].cumsum()
print(L)

plt.clf()
plt.step(L[:, 0], L[:, 1], where='post', color='k')
plt.plot(A[1:], np.full_like(A[1:], -0.3), '^b', markeredgewidth=1)
plt.plot(D[1:], np.full_like(D[1:], -0.3), 'vr', markeredgewidth=1)
plt.savefig("figures/wait4.pdf")
```

3.5 MULTI-SERVER QUEUE

Let us now generalize the simulation to a $G/G/c$ queue, i.e., there are are $c \geq 1$ servers present to serve jobs. However, before doing that, we approximate the $G/G/c$ queueing process by a $G/G/1$ queue in which the server works c times as fast. Like this we can reuse the code above to approximate a $G/G/c$ queue.

3.5.1 *A single fast server*

While you are *still in queue* of a multi-server queue, the rate at which jobs are served is the same whether there are c servers or just one server working at a rate of c. As a first simple case, we model the multi-server queueing system as if there is one fast server working at rate c.

Ex 3.5.1. Explain that we can implement a fast server by changing the service times as follows:

Python Code

```
c = 3
S = np.random.exponential(scale=1 / (c * mu), size=len(A))
```

Here is the same idea, but the implementation is slightly different. This is useful because we will see that we can generalize this to a multi-server queue in which the servers have different speeds.

Python Code

```
c = 3
W = np.zeros_like(S)
for k in range(len(S)):
    W[k] = max(W[k - 1] + S[k - 1]/c - X[k], 0)
```

3.5.2 *A real multi-server queue*

Here is the code to implement the $G/G/2$ queue; see the queueing book to understand the algorithm. (I include the print statements so that you can see step by step how it works.)

There is one very convenient feature of numpy, which is used in the `this` line of the $G/G/2$ queue code below. The feature is called *broadcasting* and is best explained with an example.

Ex 3.5.2. Run this code and explain what it does.

Python Code

```
import numpy as np

a = np.array([2, 3, 4])
a = a - 1 # or a -= 1
print(a)
```

This is the code for the $G/G/2$ queue.

Python Code

```
import numpy as np

num = 5

X = np.ones(num + 1)
S = 5 * np.ones(num)

c = np.array([1.0, 1.0])
W = np.zeros_like(S)  # store the waiting time as seen by the kth job
w = np.zeros_like(c)  # waiting times at each of the servers
for k in range(1, len(S)):
    s = w.argmin()  # server with smallest waiting time
    W[k] = w[s]
    print(f"{k=}", w, W[k])
    w[s] += S[k]
    print(w)
    print(w - X[k + 1])
    w = np.maximum(w - X[k + 1], 0)  # this
    print(w)

print(W)
```

Ex 3.5.3. Run the code, and explan the results.

As as a test we could set the vector of server capacities like `c=[1]` so that we reduce our multi-server queue to a single-server queue, and compare the output with our earlier simulators. BTW: such 'dumb' corner cases are necessary to test code. In fact, it has happened many times that I tested code of which I was convinced it was correct, but I still managed to make bugs. A bit of paranoia is a good state of mind when it comes to coding.

Now that we have tested the implementation (in part), here is the code for a queue served by three servers, but they can work at different speeds.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 1.1
N = 1000

X = np.random.exponential(scale=1 / labda, size=N + 1)
S = np.random.exponential(scale=1 / mu, size=N)

c = np.array([1.0, 1.0, 1.0])
W = np.zeros_like(S)
w = np.zeros_like(c)
for k in range(len(S)):
    s = w.argmin()
    W[k] = w[s]
    w[s] += S[k] / c[s]
    w = np.maximum(w - X[k + 1], 0)

print(W.mean(), W.std())
```

Ex 3.5.4. Where in the code do we handle the fact that servers can work at different speeds?

Ex 3.5.5. Compare $\mathsf{E}[W]$ for two cases. The first is a $G/G/1$ queue with a single fast server working at rate $c = 3$. The second is a model with three servers each working at rate 1. Include your numerical results and discuss the differences.

Ex 3.5.6. Change the code for the multi-server such that the individual servers have different speeds but the total service capacity remains the same. What is the impact on $\mathsf{E}[W]$ and $\mathsf{V}[W]$ as compared to the case with equal servers? Include your numerical results.

Ex 3.5.7. Once you researched the previous exercise, provide some consultancy advice. Is it better to have one fast server and several slow ones, or is it better to have 3 equal servers? What gives the smallest average queueing time and variance? If the variance is larger when the service rates in the multi-server queue are different rates, explain the effects based on the intuition you can obtain from Sakasegawa's formula.

4 SAKASEGAWA'S FORMULA AND BATCHING RULES

4.1 INTRO

Here we study how to apply Sakasegawa's formula in various cases with and without batching, and check the quality of this approximation.

4.2 VARIOUS MODELS COMPARED

We have (at least) three different types of models for a queueing system: simulation in discrete time, simulation in continuous, and Sakasegawa's formula to compute the expected waiting time in queue. Let's see how these models compare.

4.2.1 *Discrete time simulation*

We have a machine that can serve c_k jobs on day k. When the period length is T, then $c_k \sim \text{Pois}(\mu T)$. The number jobs that arrive on k is $a_k \sim \text{Pois}(\lambda T)$. The arrivals in period k cannot be served on day k. The code to simulate this should be familiar to you by now.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 1.2 * labda
T = 8  # period length, 8 hours in a day

num = 1000

a = np.random.poisson(labda * T, size=num)
c = np.random.poisson(mu * T, size=num)
L = np.zeros_like(a, dtype=int)

L[0] = 5
for i in range(1, num):
    d = min(c[i], L[i - 1])
    L[i] = L[i - 1] + a[i] - d


print(L.mean(), L.std())
```

Ex 4.2.1. 1. Why is the expected number of arrivals in a period equal to 24?

2. Run this code (and write down the results).
3. Then change the code such that the arrivals can be served on the day they arrive. Rerun and compare the results to the case in which arrivals cannot be served on the day in which they arrive.

Ex 4.2.2. Now change the period time, which was 8 hours in a day, to $T = 1$ (so that we concentrate on what happens during an hour instead of a day).

1. Why are now the expected number of arrivals during a period equal to 3?
2. Rerun the code with and without the arrivals being served on the period of arrival.
3. compare to the previous exercise. You should notice that there is less difference in $\mathsf{E}[L]$ between the models in which you serve or don't serve jobs in the period they arrive. But why is that so?

4.2.2 *Continuous time simulation*

Here is the code.

Python Code

```
import numpy as np

np.random.seed(3)

labda = 3
mu = 1.2 * labda
num = 1000
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)

for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]

J = D - A
EL = labda * J.mean()
print(EL)
```

Ex 4.2.3. Explain that we use Little's law to compute $\mathsf{E}[L]$. Can we use this law to estimate $\mathsf{V}[L]$?

Ex 4.2.4. Run the code and compare with the discrete time simulation.

4.2.3 *Sakasegawa's formula*

Now we use Sakasegawa's formula, rather than simulation, to compute $\mathsf{E}[L]$.

Python Code

```
import numpy as np


def sakasegawa(labda, ES, Ca2=1, Cs2=1, c=1):
    rho = labda * ES
```

```
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T


labda = 3
mu = 1.2 * labda
ES = 1 / mu
W = sakasegawa(labda, 1 / mu, 1, 1, 1)
L = labda * (W + ES)
print(L)
```

Ex 4.2.5. Run the code, and compare the value with the discrete and continuous time simulations.

You should notice that there is quite a difference between the estimates for $\mathsf{E}[L]$ we obtain from simulation, on the one hand, and from Sakasegawa's formula, on the other hand. We conclude that it takes quite a bit of time for a simulation to reach 'steady state'.

4.3 SERVER SETUP AND BATCHING

We can now setup a model in which jobs arrive in batches of size B and in between batches the server needs a constant setup time R. Check the queueing book for further background; we are going to build the model of the related section.

4.3.1 *Sakasegawa's formula*

I build up the code in small blocks. You should put the code blocks underneath each other as you progress.

Since we add setup times, we should be carefull to avoid a situation which the load is too large (recall, setup times add to the service times).

Python Code

```
import numpy as np

np.random.seed(3)

B = 13
labda = 3
mu = 2 * labda
R = 2

rho = labda * (1 / mu + R / B)
assert rho < 1, f"{rho=} >= 1"
```

Ex 4.3.1.

1. What does the `assert` statement do?
2. Why do I put it here, i.e., before doing any other work?
3. What happens if you would set `labda=10` and `mu=3`.

Next, we need to get the parameters correct for the batches. I just follow the book.

Python Code

```
W1 = (B - 1) / 2 / labda
```

Ex 4.3.2. What is `W1` conceptually:?

For the queueing time, we have this:

Python Code

```
labdaB = labda / B
VR = 0  # constant R
ca2 = 1 / B
ES0 = 1 / mu
ES = ES0 + R / B
ESB = B * ES
VS0 = ES0 * ES0
VSB = VR + B * VS0
cs2 = VSB / ESB ** 2
```

Ex 4.3.3. If the interarrival times $X_k \sim \text{Exp}(\lambda)$, why should we set `ca2 = 1/B`?

For the average queueing time:

Python Code

```
def sakasegawa(labda, ES, Ca2, Cs2, c=1):
    rho = labda * ES
    V = (Ca2 + Cs2) / 2
    U = rho ** (np.sqrt(2 * (c + 1)) - 1) / (1 - rho)
    T = ES / c
    return V * U * T

W2 = sakasegawa(labdaB, ESB, ca2, cs2)
```

Ex 4.3.4. Why should we call the function `sakasegawa` with the following parameters `sakasegawa(labdaB, ESB, ca2, cs2)`?

The last step of the queueing process can be coded like this:

Python Code

```
W3 = R + (B + 1) / 2 * ES0
```

Ex 4.3.5. What is the meaning of `W3`?

The sojourn time:

Python Code

```
J = W1 + W2 + W3
print(J)
```

4.3.2 *Simulation*

To set up the simulation requires a bit fiddling with slicing. It took me a bit of time, and print statements, to get the details right. Here is the code, with the print statements so that you can figure out how it works.

Python Code

```
num = 1000
num = B * (num // B)  # get multiple of B
X = np.random.exponential(scale=1 / labda, size=num)
A0 = np.zeros(len(X) + 1)
```

```
A0[1:] = X.cumsum()
A = np.zeros_like(A0)
for i in range(num // B):
    st = i * B + 1  # start
    fi = (i + 1) * B  # finish
    A[st : fi + 1] = A0[fi]


S0 = np.random.exponential(scale=1 / mu, size=len(A))
S0[0] = 0
S = S0.copy()
S[1::B] += R

D = np.zeros_like(A)
for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]

J = D - A0
print(J.mean())
```

Ex 4.3.6. Clearly, `A0` are the arrival times of the jobs at the system. Explain that `A` corresponds to the arrival times of the *batches* at the queue, and that `S` are the service times including setup times. (Use print statements to understand how the slicing, i.e., notation like `A[st : fi +1]`, works.)

Ex 4.3.7. Why is `D-A0` the sojourn time, and not `D - A`?

Ex 4.3.8. Run the code for `num = 1000` and compare the results of the formulas and the simulation. (Ensure that both models use the same data.) Then extend to `num = 1_000_000` and check again. What do you see, and conclude?

4.3.3 *Getting things right*

While making the above code, I made several (tens of) errors, so that the simulation and the formulas gave different results. Here are the steps that I followed to get things right. Only after one step was correct, I moved on to the next.

1. Check with $B = 1$ and $R = 0$, since $B = 1$ is the single job case.
2. Keep $B = 1$, set $R = 0.2$. I had to change μ so that still ρ 1.
3. Set $B = 2$, $R = 0$. Compare `ES` (input for Sakasegawa's formula) to `S.mean()` (input for simulation).
4. In the previous step I did not get corresponding results for `num = 10000`. Changing it to 1 million helped.

After these four steps, the simulation and the model gave similar results. However, from a higher level of abstraction, I am not quite happy about this. It is not realistic to wait until we have seen a million or so arrivals in any practical setting. My personal way to deal with this situation is like this (but not all people agree on what to do though):

- Simple formulas are tremendously useful to get *insight* into the main drivers of the behavior of a system. In other words, there is not better way to get *qualitative* understanding than with simple formulas.

- The quantitative quality of a formula need to not be too good.
- Building a simulator is intellectually rewarding as it helps understand the *dynamics* of a system.
- Building a simulator is difficult; it's easy to make mistakes, in the code, in the model, in the data...
- Simulation depends on large quantities of data. It's very hard (next to impossible) to *understand* the output.
- The simple formulas can be used to check the output of a simulator when applied to simple cases.

All in all, I think that simulation and theoretical models should go hand in hand, as they offer different type of insight, and have different strengths and weaknesses.

4.4 SERVER ADJUSTMENTS

Consider now a queueing system in which the server needs an adjustments with probability p (see the section on server adjustments in the book). The repair times are assumed constant, at first, with mean 2. I expect you now to be able to fill in Sakasegawa's formula. It remains to show you how to simulate this system.

Python Code

```
import numpy as np

np.random.seed(3)

labda = 3
mu = 2 * labda
p = 1 / 20
num = 10000

X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
R = 2 * np.ones(len(S))                                  # this
I = np.random.uniform(size=len(S)) <= p
D = np.zeros_like(S)

for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k] + R[k] * I[k]

W = D - A - S
print(W.mean())
```

Ex 4.4.1. Explain how `D` is computed.

Ex 4.4.2. To test the code I set at first `R = 0 * np.ones(len(A))` in the line marked as `this`. Why would I do this (to what simpler queueing system can I compare the results of this program?)

Ex 4.4.3. Now run the code, with `R` as in the code (not set as 0 such as in the previous exercise). Compare the answers. Then set `num = 100000`, i.e., 10 times as large. What is the effect?

Ex 4.4.4. Now set `R = np.random.exponential(scale=2, size=len(A))`. What is the effect on $\mathsf{E}[W]$? In general, do you see that indeed $\mathsf{E}[W]$ increases with the variability of the adjustments?

4.5 SERVER FAILURES

This time we focus on a server that can fail; again check the queueing book for the formulas. Here we just implement them.

Python Code

```
import numpy as np
from scipy.stats import expon

np.random.seed(3)

labda = 2
mu = 6
ES0 = 1 / mu
labda_f = 2
ER = 0.5
num = 10000

X = np.random.exponential(scale=1 / labda, size=num)
X[0] = 0
A = X.cumsum()

S0 = np.random.exponential(scale=ES0, size=num)
N = np.random.poisson(labda_f * ES0, len(S0))
R = expon(scale=ER)
S = np.zeros_like(S0)
for i in range(len(S0)):
    S[i] = S0[i] + R.rvs(N[i]).sum()


D = np.zeros_like(A)
for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]

W = D - A - S
print(W.mean())
```

Ex 4.5.1. Run the code and include the results in your assignment. Where do we include the break downs?

Ex 4.5.2. Suppose you can choose between two alternative ways to improve the system. Increase λ_f, and decrease $\mathsf{E}[R]$, but such that $\lambda_f \mathsf{E}[R]$ remains constant; or the other way around, decrease λ_f and increase $\mathsf{E}[R]$. Which alternative has better influence on $\mathsf{E}[W]$?

4.6 A SIMPLE TANDEM NETWORK

We have two queues in tandem. Here is the code to simulate this.

Python Code

```
import numpy as np

```

```
np.random.seed(4)

labda = 3
mu1 = 1.2 * labda
num = 100000
X = np.random.exponential(scale=1 / labda, size=num)
A1 = np.zeros(len(X) + 1)
A1[1:] = X.cumsum()
S1 = np.random.exponential(scale=1 / mu1, size=len(A1))
D1 = np.zeros_like(A1)

for k in range(1, len(A1)):
    D1[k] = max(D1[k - 1], A1[k]) + S1[k]

W1 = D1 - A1 - S1

# queue two
A2 = D1
mu2 = 1.1 * labda
S2 = np.random.exponential(scale=1 / mu2, size=len(A2))
D2 = np.zeros_like(A2)

for k in range(1, len(A2)):
    D2[k] = max(D2[k - 1], A2[k]) + S2[k]

W2 = D2 - A2 - S2

J = D2 - A1
print(W1.mean(), S1.mean(), W2.mean(), S2.mean(), J.mean())
```

Ex 4.6.1. Why is `J = D2 - A1`?

5 CONTROL OF QUEUES

5.1 INTRO

We simulate queues that are controlled by some policy that uses information on waiting time or queue length. We also develop algorithms to compute the state probabilities of the $M/G/1$ queue under the policies.

5.2 BLOCKING ON WAITING TIME

In this assignment we investigate queues under certain control rules. First we focus on blocking, then on switching on and off the server depending on the queue length.

Suppose we don't allow jobs to enter the system when the waiting time becomes too long. A simple rule is the block job k when $W_k \geq T$, for some threshold T.

How to simulate that?

5.2.1 *A start with no blocking*

Before doing something difficult, I tend to start from a situation that I do understand, which, in this case, is the single server queue without blocking.

Here is our standard code. With this we can find parameters that are suitable to see that blocking will have an impact. (Suppose $\lambda = 1$, $\mu = 1000$, $T = 100$, we will see no job being blocked during any simulation, at least not in this universe.)

Python Code

```
import numpy as np


np.random.seed(3)

num = 10000
labda = 1
mu = 1.1 * labda
T = 5
X = np.random.exponential(scale=1 / labda, size=num)
S = np.random.exponential(scale=1 / mu, size=len(X))
S[0] = 0

W = np.zeros_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)

print(W.mean(), W.max())
print((W >= T).mean()) # this
```

Ex 5.2.1. Run the code. What is the purpose of the `this` line?

5.2.2 *A hack to implement blocking*

Here is a dirty hack to implement blocking. When $W_k \geq T$, job k should not enter. That means that its service time should not be added to the waiting time. But not adding the service time can be achieved by setting $S_k = 0$. To achieve this, we can change the for loop as follows.

Python Code

```
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)
    if W[k] >= T:
        S[k] = 0
```

Ex 5.2.2. Run this code and check its effect on $\mathsf{E}[W]$, $\mathsf{V}[W]$, and $\max\{W_k\}$.

5.2.3 *A better method*

As a matter of principle, I don't like the code of the previous section. In my opinion such hacks are a guarantee on bugs that can be very hard to find later. Mind, with this trick I am changing my primary data, in this case the service times. Reuse these service times at a later point in the code, for instance for a comparison with other models or for testing, has become impossible. And if I forget this (when I use this code maybe half a year later), then finding the bug will be very hard. Hence, as a golden rule: don't touch the primary data.

Here is better code.

Python Code

```
W = np.zeros_like(S)
I = np.ones_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] * I[k - 1] - X[k], 0)
    if W[k] >= T:
        I[k] = 0

print(W.mean(), I.mean())
```

Ex 5.2.3. What does the vector `I` represent?

5.2.4 *Some other blocking rules*

There are other rules to block jobs.

Ex 5.2.4. In this code, what is the rule to block jobs?

Python Code

```
W = np.zeros_like(S)
I = np.ones_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] * I[k - 1] - X[k], 0)
    if W[k] + S[k] >= T:
        I[k] = 0

print(W.mean(), W.max(), I.mean())
```

Ex 5.2.5. Likewise, how does this rule work? What is the meaning of `V`?

Python Code

```
W = np.zeros_like(S)
V = np.ones_like(S)
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + V[k - 1] - X[k], 0)
    V[k] = min(T - W[k], S[k])

print(W.mean(), W.max(), S.mean() - V.mean())
```

5.3 BATCH QUEUES AND BLOCKING ON WAITING TIME

Let us now set up a simulation to see the combined effect of batch arrivals and blocking on waiting time.

Recall, in the queueing book we discuss some methods to block jobs in the $M^X/M/1$ queue when the queue length (not the waiting time) is too long. We tackle blocking on queue length in a separate section below.

5.3.1 *Again start without blocking*

We need a slightly different way to generate service times. When a batch of B_k jobs arrives at time A_k, then the service time added to the waiting is the sum of the service times of all B_k jobs in the batch.

Python Code

```
import numpy as np
from scipy.stats import expon

np.random.seed(3)

num = 10000
labda = 1
mu = 2 * labda
X = np.random.exponential(scale=1 / labda, size=num)
B = np.random.randint(1, 2, size=num)
S = expon(scale=1 / mu)

W = np.zeros_like(X)
for k in range(1, len(W)):
    W[k] = max(W[k - 1] + S.rvs(B[k]).sum() - X[k], 0)

ES = 1/mu
rho = labda * ES
print(S.mean(), 1 / mu)
print(rho / (1 - rho) * ES, W.mean(), W.max())
```

Ex 5.3.1. Explain how this code works.

Ex 5.3.2. Run the code. Why do I take `B` as it is here (recall, I like to test)? Why should `W.mean()` and $\rho\,\mathsf{E}[S]/(1-\rho)$ be approximately equal?

The next remark applies only if you're interested, otherwise skip. `numpy.random` provides functionality to generate random numbers, but not more, while `scipy.stats` provides much more useful functions. For the above, observe that `expon` loads from `scipy.stats`, while the other rvs come from `numpy.random`. Check the manuals on the web for further information.

5.3.2 *Include blocking*

Here is the code with a blocking rule.

Python Code

```
import numpy as np
from scipy.stats import expon

np.random.seed(3)

num = 1000
labda = 1
mu = 3.1 * labda
T = 5
X = np.random.exponential(scale=1 / labda, size=num)
B = np.random.randint(1, 5, size=num)
S = expon(scale=1 / mu)

W = np.zeros_like(X)
V = np.zeros_like(W)
for k in range(1, len(W)):
    W[k] = max(W[k - 1] + V[k - 1] - X[k], 0)
    V[k] = S.rvs(B[k]).sum() if W[k] < T else 0

print(S.mean() * B.mean() - V.mean())
print(W.mean(), W.max())
print(np.isclose(V, 0).mean())
print((V <= 0).mean())  # this
```

Ex 5.3.3. 1. Explain how the code works.

2. What do the printed KPIs mean?

3. Finally, in the `this` line, why is it better to use `np.isclose` instead?

5.4 BLOCKING ON QUEUE LENGTH

Blocking on queue length is quite a bit harder with a simulation in continuous time because we need to keep track of the number of jobs in the system. (Recall in discrete time the recursions to compute $\{L_k\}$ are easy, while in continuous time the recursions for $\{W_k\}$ or $\{J_k\}$ are easy.)

5.4.1 *Start without blocking*

As before, I start from a code that I really understand, and then I extend it to a situation that I find more difficult. So, here is code to find the system length L at *arrival* epochs $\{A_k\}$.

Python Code

```
import numpy as np

```

```
np.random.seed(3)

num = 10000
labda = 1
mu = 1.5 * labda
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)
L = np.zeros_like(A, dtype=int)

idx = 0
for k in range(1, len(A)):
    D[k] = max(D[k - 1], A[k]) + S[k]
    while D[idx] < A[k]:
        idx += 1
    L[k] = k - idx

rho = labda / mu
print(L.mean(), rho/(1-rho), L.max())
print((L == 0).mean(), 1 - rho)
print((L == 1).mean(), (1 - rho)*rho)
```

Ex 5.4.1. Explain how this computes `L[k]`. Do we count the system length as seen upon arrival, or does `L[k]` include job k, i.e., the job that just arrived?

Ex 5.4.2. Just to check that you really understand: why is it ok here to use `(L == 0)` rather than `np.close`?

Ex 5.4.3. Why do I compare `L.mean()` to $\rho/(1-\rho)$ and not to $\rho^2/1-\rho$)?

Ex 5.4.4. *<2022-03-16 wo>* Please skip this exercise. Change μ to 1.05λ. Now the results of the simulation are not very good if `num=1000` or so. Making `num` much larger does the job, though.

5.4.2 *Include blocking*

It might seem that we are now ready to implement a continuous time queueing system with blocking on the queue length. Why not merge the ideas we developed above? Well, because this does not work.

(If you like a challenge, stop reading here, and try to see how far you can get with developing a simulation for this situation.)

Only after having worked for 3 hours I finally saw 'the light'. As a matter of fact, I needed a new data structure, a `deque` from which we can pop and append jobs at either end of a list. Here is the code.

Python Code

```
from collections import deque
import numpy as np

np.random.seed(3)

```

```
num = 10000
labda = 1
mu = 1.2 * labda
T = 5
X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
S[0] = 0
D = np.zeros_like(A)
L = np.zeros_like(A, dtype=int)

Q = deque(maxlen=T + 1)
for k in range(1, len(A)):
    while Q and D[Q[0]] < A[k]:
        Q.popleft()
    L[k] = len(Q)
    if len(Q) == 0:
        D[k] = A[k] + S[k]
        Q.append(k)
    elif len(Q) < T:
        D[k] = D[Q[-1]] + S[k]
        Q.append(k)
    else:
        D[k] = A[k]
```

Ex 5.4.5. Read the documentation of how a `deque` works, then explain the code.

Ex 5.4.6. What queueing discipline would result if we would use the `pop()` and `appendleft()` methods of a `deque`?

Ex 5.4.7. What queueing discipline would result if we would use the `pop()` and `append()` methods of a `deque`?

Ex 5.4.8. Run this code with `T=100` and compare this with the queueing system without blocking. Why should you get the same results? (Realize that this is a check on the correctness of our code.)

Glue the next code (for the theoretical model) at the end of the previous code.

Python Code

```
rho = labda / mu
p = np.ones(T + 1)
for i in range(1, T + 1):
    p[i] = rho * p[i - 1]
p /= p.sum()
for i in range(T + 1):
    print((L == i).mean(), p[i])
```

Ex 5.4.9. Now set `T=5` and `num = 10000` or so. Run the code. Why do the result agree with the theoretical model? Why is this the $M/M/1/T$ queue?

In fact, I used the above theoretical model to check whether the simulation was correct. (My first 20 or so attempts weren't.)

6 SIMULATION WITH EVENT STACKS

6.1 GENERAL INFO

The tools we developed up to now do not suffice to simulate more difficult (queueing) systems. For harder cases we need a concept called *event stack*. Here we show how to implement an event stack as a *heap queue*. Finally, we will use python *classes* as these are very natural objects to use in complicated simulations. At the end you will see how general and clean our final simulation environment is.

6.2 SORTING WITH HEAP QUEUES

We start with a simple problem: how to sort a bunch of students by age? For this we will use a *heap queue*. This is a stack on which we can push elements, then, when popping from the stack, the elements come out in a sorted way.

Ex 6.2.1. To understand the code below, first look up on the web:

1. what is the difference between a *tuple* and a *list*?
2. When to use one or the other?

As always, keep your answer brief, just show that you understand; that suffices.

The heap in our code below uses the first element of the *tuple* of a student's age and name to sort the students. Let's put a number of students on the stack. We need a new modules `heapq`; this provides the algoritms for pushing to and popping from a stack.

Python Code

```
from heapq import heappop, heappush

stack = []

heappush(stack, (25, "Cynthia"))
heappush(stack, (21, "James"))
heappush(stack, (21, "James"))
heappush(stack, (20, "Pete"))
heappush(stack, (18, "Clair"))
heappush(stack, (14, "Jim"))
print(stack)
```

Observe that with popping we take the element from the stack with the smallest age. You should know that anytime we pop from the stack, the stack becomes one element shorter. (And with pushing it becomes longer.)

Python Code

```
age, name = heappop(stack)
print(age, name)
```

Ex 6.2.2. Add the following lines to the code above, and include the output. You should see that the students are printed in order of age.

Python Code

```
while stack:
    e = heappop(stack)
    print(e)
```

6.3 CLASSES

In a class we can organize information.

Python Code

```
class Student:
    def __init__(self, name, age, phone):
        self.name = name
        self.age = age
        self.phone = phone

    def __repr__(self):
        return f"{self.name}, {self.age}, {self.phone}"

```

Ex 6.3.1. Look up on the web: what does the `repr` method do?

Making an *object* of a *class* is called *instantiation*.

Python Code

```
hank = Student("Hank", "21", "Huawei")
print(hank)
```

Let's add some more students and put them in a list.

Python Code

```
students = [
    Student("Joseph", "18", "Motorola"),
    Student("Maria", "21", "Huawei"),
    Student("Natasha", "20", "Apple"),
    Student("Chris", "25", "Nexus"),
]
```

Ex 6.3.2. Add two more students, e.g., you and your group mate, to the list. (And if you don't like to spread such details, just lie about your age :-)) Show your code.

With heaps we can sort the students in any sequence we like. Let's sort them by phone brand.

Python Code

```
from heapq import heappop, heappush


class Student:
    def __init__(self, name, age, phone):
        self.name = name
        self.age = age
        self.phone = phone

```

```
    def __repr__(self):
        return f"{self.name}, {self.age}, {self.phone}"


students = [
    Student("Joseph", "18", "Motorola"),
    Student("Maria", "21", "Huawei"),
    Student("Natasha", "20", "Apple"),
    Student("Chris", "25", "Nexus"),
]

stack = []

for s in students:
    heappush(stack, (s.phone, s))

while stack:
    print(heappop(stack))
```

Ex 6.3.3. What line makes that the students are sorted by phone brand?

Ex 6.3.4. Change the code so that you can sort the students by age. Show the line(s) of your code to achieve this.

Ex 6.3.5. Extend the `Job` class such that we can give the student also a surname. Then invent some surnames, and sort the students by surname. Include your code to show how you did this.

6.4 A JOB CLASS USEFUL FOR SIMULATION

How can use heap queues in the simulation of queueing systems? To see this, think of time as a sequence of events in which things happen. In a queueing system, three things can happen: a job arrives, a job service starts, or a job leaves. For a job we should specify its arrival and departure times. We store the jobs in a heap queue and add the arrival time as a key to pop jobs in order of arrival.

We store the arrival, service, and departure time as the *attributes* of a job class. We also store the queue length at arrival times to gather statistics once the simulation is over.

Python Code

```
from heapq import heappop, heappush

ARRIVAL, DEPARTURE = 0, 1


class Job:
    def __init__(self):
        self.arrival_time = 0
        self.service_time = 0
        self.departure_time = 0
        self.queue_length_at_arrival = 0

    def sojourn_time(self):
        return self.departure_time - self.arrival_time

    def waiting_time(self):
```

```
17          return self.sojourn_time() - self.service_time
18
19      def service_start(self):
20          return self.departure_time - self.service_time
21
22      def __repr__(self):
23          return f"{self.arrival_time}, {self.service_time}, {self.service_start()}, \
24                   {self.departure_time}\n"
25
26      def __le__(self, other):
27          # this is necessary to sort jobs when they have the same arrival times.
28          return self.id <= other.id
29
30
31  events = []  # event stack, global
32  num_jobs = 5
33  interarrival_time = 3
34  service_time = 5
35
36  arrival_time = 0
37  for i in range(num_jobs):
38      arrival_time += interarrival_time
39      job = Job()
40      job.arrival_time = arrival_time
41      job.service_time = service_time
42      heappush(events, (job.arrival_time, job, ARRIVAL))
43
44  while events:
45      time, job, typ = heappop(events)
46      print(job)
```

Ex 6.4.1. Explain the waiting time and sojourn functions of the `Job` class.

For later purposes, we have to add labels to the events in the heap queue to indicate what type of event we are dealing with, an arrival or a departure.

Ex 6.4.2. Change the `interarrival_time` to some number you like. Run the code, include your results, and explain the output of the print statement.

6.5 A VERY POWERFUL GG1 SERVER SIMULATOR

The code here below is a MWE (minimal working example). You should copy it and run it. Below we'll ask questions in the exercises about how the code works.

6.5.1 *Start*

Copy this first.

Python Code

```
1  from itertools import islice
2  from heapq import heappop, heappush
3  import numpy as np
4  from scipy.stats import expon, uniform
5
6  import matplotlib.pylab as plt
7
```

```
8  np.random.seed(3)
9
10
11 ARRIVAL, DEPARTURE = 0, 1
12 IDLE, BUSY = 0, 1
```

Put the code for `Job` class after this.

6.5.2 *GG1 simulator.*

Copy `GG1` class and put this af the `Job` class. We put the code on one page so that the copying is easy.

Python Code

```
class GG1:
    def __init__(self, F, G, num_jobs):
        self.F = F  # interarrival time distribution
        self.G = G  # service time distribution
        self.num_jobs = num_jobs
        self.queue = []
        self.served_jobs = []  # assemble statistics
        self.state = IDLE

    def make_jobs(self):
        time = 0
        for i in range(num_jobs):
            time += self.F.rvs()
            job = Job()
            job.arrival_time = time
            job.service_time = self.G.rvs()
            heappush(events, (job.arrival_time, job, ARRIVAL))

    def run(self):
        while events:  # not empty
            time, job, typ = heappop(events)

            if typ == ARRIVAL:
                self.handle_arrival(time, job)
            else:
                self.handle_departure(time, job)

    def handle_arrival(self, time, job):
        job.queue_length_at_arrival = len(self.queue)
        if self.state == IDLE:
            self.state = BUSY
            self.start_service(time, job)
        else:
            self.put_job_in_queue(job)

    def start_service(self, time, job):
        job.departure_time = time + job.service_time
        heappush(events, (job.departure_time, job, DEPARTURE))

    def put_job_in_queue(self, job):
        heappush(self.queue, (job.arrival_time, job))

    def handle_departure(self, time, job):
        if self.queue:  # not empty
            _, next_job = heappop(self.queue)
            self.start_service(time, next_job)
        else:
            self.state = IDLE
        self.served_jobs.append(job)

```

Ex 6.5.1. Explain how the following methods work:

1. arrivals: `GG1.handle_arrival()`;
2. departures: `GG1.handle_departure`;
3. running the simulation: `GG1.run`.

You should observe that, with a heap queue, we can let the heap queue do all the work of tracking time. It is not easy to understand how we use a queue (that is, the heap queue) to simulate queueing systems. However, once you understand how event stacks work, you know all there is to discrete event simulation.

6.5.3 *Running the simulator*

Put this code the `GG1` class.

Python Code

```
labda = 2.0
mu = 3.0
rho = labda / mu
F = expon(scale=1.0 / labda)  # interarrival time distribution
G = expon(scale=1.0 / mu)  # service time distribution
num_jobs = 100

events = []

gg1 = GG1(F, G, num_jobs)
gg1.make_jobs()

print(list(islice(events, 0, 5)))
```

Ex 6.5.2. To view the contents of the first couple of events I tried this: `print(events[:5])`, but that failed. After a bit of searching on the web I found that I had to use `islice`.

1. Read on the web what `islice` does, and explain it in your own words.
2. Explain also why we need to turn the output of `islice` into a `list`. (This requires that you read and think about what a generator is.)

Ex 6.5.3. Explain the results of the line `print(list(islice(events, 0, 5)))`. Why is the content of column 5 negative?

6.5.4 *Analyze the results, statistics*

Put this at the end of the code.

Python Code

```
gg1.run()

sojourn = np.zeros(len(gg1.served_jobs))
for i, job in enumerate(gg1.served_jobs):
    sojourn[i] = job.sojourn_time()

print(sojourn.mean(), sojourn.std(), sojourn.max())

plt.clf()
plt.plot(sojourn)
plt.savefig('figures/sojourn0.png')
```

Ex 6.5.4. Change the seed of the random number generator, choose your favorite number of jobs (something positive, reasonably small). make your own plot and include it in your document. Include also the statistics such as the mean.

If we would use this code for real world problems, we of course should compare the results of this simulator to our earlier simulators; for the assignment we skip this.

6.6 OTHER SCHEDULING RULES

Suppose we prefer to serve the shortest job in queue. We can inherit all of our `GG1` queue, except the scheduling rule. For this we need to change just one line of the class. The implementation of other scheduling is also extremely easy now

6.6.1 *SPTF scheduling*

Here is a class to simulate the SPTF (shortest processing time first). We can derive all from `GG1` except that we have to change the rule on how to put the jobs on the queue.

Python Code

```
class SPTF_queue(GG1):
    def put_job_in_queue(self, job):
        heappush(self.queue, (job.service_time, job))
```

Ex 6.6.1. Explain how this code implements SPTF.

Ex 6.6.2. To run it, use this code. Include the results.

Python Code

```
sptf = SPTF_queue(F, G, num_jobs)
sptf.make_jobs()
sptf.run()
print(sptf.served_jobs[:5])
```

Ex 6.6.3. If you're interested (but you can skip this) print at the end the average number of jobs in the system and compare that to $\mathsf{E}[L]$ for the FIFO queue. What difference do you see?

6.6.2 *LIFO scheduling*

Last-in-First-Out is also trivial.

Python Code

```
class LIFO_queue(GG1):
    def put_job_in_queue(self, job):
        heappush(self.queue, (-job.arrival_time, job))
```

Ex 6.6.4. Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times. Comment on your findings.

6.6.3 *Serve longest job first*

Ex 6.6.5. Update the code of the `SPTF` queue such that the longest job is selected from the queue, rather than the shortest. Include the relevant lines of the code, and explain why it implements the SLJF rule.

Hopefully you see how easy it is (now we have done all the hard work) to compare different job scheduling rules. For instance, for the earliest due data first rule, we add a due date attribute to each job, and when selecting a job for service we choose the earliest due date.

7 HINTS

h.1.3.3. What is the average number of arrivals per period? What is the average number of jobs that can be served, i.e., the average service capacity? Are they close or not?

h.1.3.8. As a simple analogous problem: imagine you have bucket containing 10 liters. Water flows in from a hose at rate 3 liters per minute, but it flows out via another hose at rate 5 l/m. What is the net outflow? Why does it take 5 minutes before the bucket is empty?

h.1.3.10. See the queueing book exercises 2.1.8 and 3.2.3 for further explanations.

When $L_0 \gg 1$, then a_k jobs arrive in period k and c_k jobs leave. For ease, write $h_k = c_k - a_k$. Then the time τ to hit 0, i.e., the time until the queue is empty, is the smallest τ such that $\sum_{k=1}^{\tau} h_k \geq L_0$. But then, by Wald's theorem: $\mathsf{E}[\tau]\,\mathsf{E}[h_k] = L_0$. What is $\mathsf{E}[\tau]$?

Moreover, consider any sum of random variables, then with $\mu = \mathsf{E}[X]$, σ the std of X, and N the normal distribution, and by the central limit theorem,

$$\frac{1}{n}\sum_{i=1}^{n} X_i \sim N(\mu, \frac{\sigma^2}{n}) \implies \sum_{i=1}^{n} X_i \sim N(n\mu, n\sigma^2). \tag{2}$$

h.3.2.3. The X_k are iid.

h.3.2.4. If I would not do this, and I would want to change the simulation length (the number of jobs), at how many places should I change this number?

h.3.2.6. Did we really serve job 0? If `num` is big number, does it matter that we set `S[0]=0`?

h.3.2.7. We subtract `S[0]` as if we served the corresponding job, but did we actually serve it?

h.3.3.1. We include `W[0]`, but what is that?

h.3.3.8. The crucial points are (A_k, W_k), $(A_k, W_k + S_k)$, and $(D_k, 0)$ when $W_{k-1} = 0$. Then connect these points with straight lines.

h.3.4.2. The improved algorithm in $O(n)$, while the previous was $O(n^2)$.

h.3.5.5. Take `c=[3.]` for the first case, and `c=[1., 1., 1.]` for the second. With this, the change in code is minimal.

h.3.5.6. For instance, set `c=np.array([2, 0.5, 0.5])`.

h.4.2.1. Replace the relevant line by `d = min(c[i], L[i - 1] + a[i])`.

h.4.3.7. Recall that $A0$ are the arrival times at the system before batching while A are the times the jobs move as a batch to the queue.

h.5.2.3. If `I[k] == 1`, then what happens to job k?

h.5.3.2. When the batches `B = np.random.randint(1, 2, size=num)`. Hence, we deal with the $M/M/1$ queue for this choice of B.

h.5.4.1. When the while loop terminates, is `idx` the index of the last departure, or does it point to the job that is the first to leave?

h.5.4.2. Is `L` a float?

h.5.4.3. What is $\rho^2/(1-\rho)$?

h.5.4.6. Does it matter whether we push jobs from right to left through a quue, rather then from left to right?

h.5.4.7. It's not FIFO.

h.5.4.8. Is `L.max()` larger than 100 for this simulation?

h.6.6.5. Put a minus sign at the right position when adding a job to the queue heap.