Queueing theory Assignments

EBB074A05

Nicky D. van Foreest

2022:02:16

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

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

Python Code

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

1.2.2 *two periods*

Python Code

```python
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

```python
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

```python
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 ρ< 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

```python
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 $\{\dots, -2, -1, 0\}$—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

```python
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

```python
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
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

```
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

```
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

```python
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

```python
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

```python
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
```python
a = np.random.poisson(11.8, 10)
print(a)
```

2.2.4 *Service capacity*

Python Code
```python
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
```python
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*

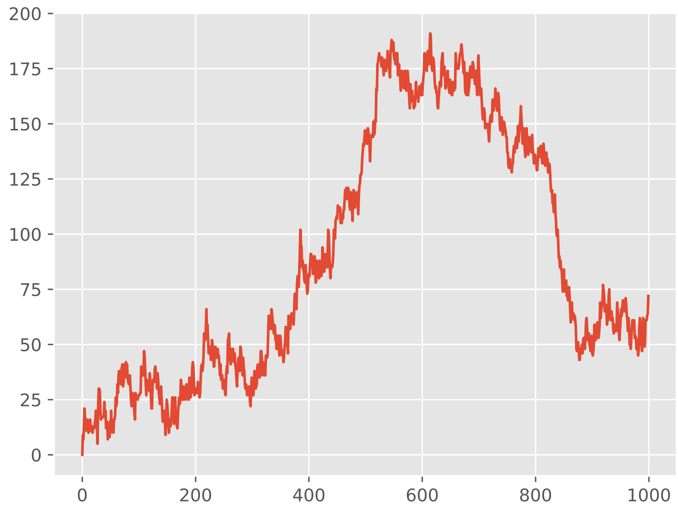
Python Code
```python
s = np.sum(p, axis=0)
print(s)
```

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")
```



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.

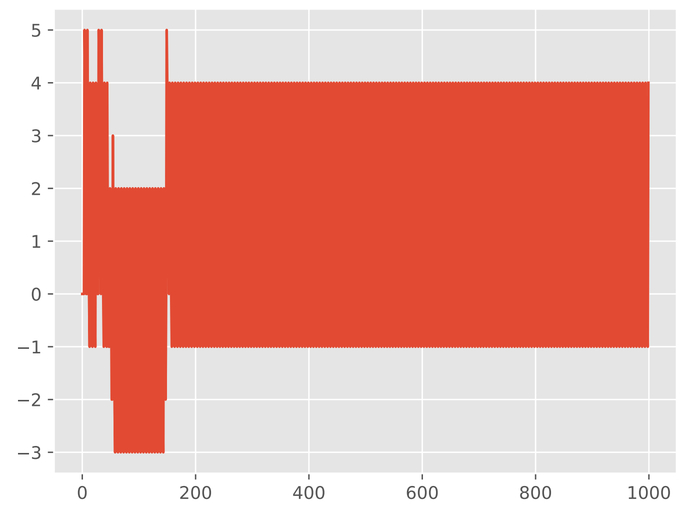
Python Code

```
def balanced(a):
    p = np.empty([5, len(a)])
    p[0, :] = 2.0 * np.ones_like(a)
    p[1, :] = 2.0 * np.ones_like(a)
    p[2, :] = 3.0 * np.ones_like(a)
    p[3, :] = 4.0 * np.ones_like(a)
    p[4, :] = 4.0 * np.ones_like(a)
    return p

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


```

```
p = balanced(a)
spread_holidays(p)
s = np.sum(p, axis=0)
Q2 = computeQ(a, s)

plt.plot(Q2)
plt.savefig("figures/psych2.pdf")
```

Python Code

```
plt.clf()
plt.plot(Q1-Q2)
plt.savefig("figures/psych22.pdf")
```



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.

2.3.2 *Synchronize holidays*

What is the effect of all psychiatrists taking holidays in the same week?

Python Code

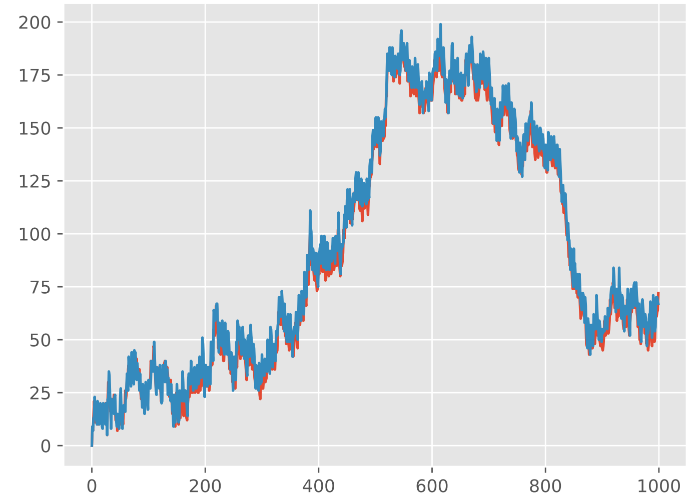
```
a = np.random.poisson(11.8, 10)


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

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

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

Ex 2.3.4. 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.

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. 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.

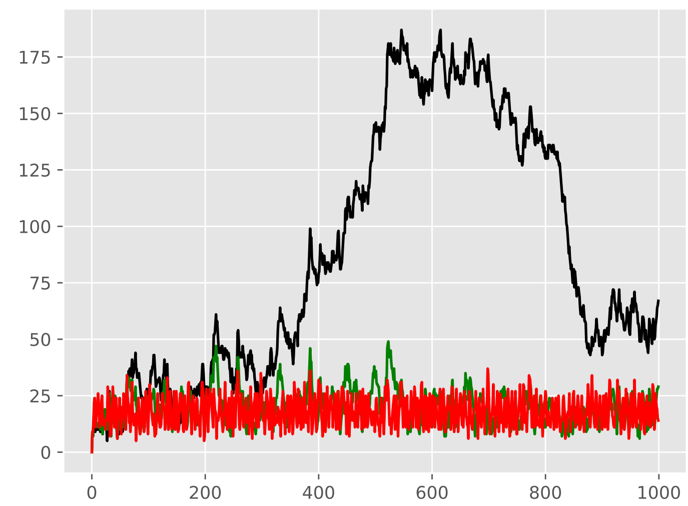
Python Code

```python
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
        else:
            C = c
        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

```python
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

```python
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 $\sim \mathrm{Exp}(\lambda)$, 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

```python
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

```python
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)
```

This is better!

Ex 3.2.3. 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)
```

Ex 3.2.7. Explain now why it is practical to have $A_0 = 0$.

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 3.2.8. 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

3.3.1 *Relevant averages*

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

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

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

Ex 3.3.1. 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
```python
rho = S.sum() / D[-1]
idle = (D[-1] - S.sum()) / D[-1]
print(idle)
```

Ex 3.3.2. 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
```python
idle_times = np.maximum(A[1:] - D[:-1], 0)
print(idle_times)
print(idle_times.sum())
print(D[-1] - S.sum())
```

Ex 3.3.3. 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 3.3.4. 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

```python
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 3.3.5. 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

```python
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 3.3.6. 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 mey 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 3.3.7. Make a plot of the virtual waiting time by hand. Figure 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

```python
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 3.3.8. Explain how we store the relevant epochs in `E`. Why do we use `idx[1:]` (What is `W[0]`?). Why do we subtract 1 from `idx[1:]`? 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 QUEUE LENGTH

We have the waiting times, but not the number of jobs in queue. What if we would like to plot the queue length process?

A simple, but inefficient, algorithm to construct the queue length process is to walk backwards in time.

Python Code

```python
import numpy as np
np.random.seed(3)

num = 10
X = np.random.exponential(scale=labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
mu = 0.8 * labda
S = np.random.exponential(scale=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)):
    l = k - 1
    while D[l] > A[k]:
        l -= 1
    L[k] = k - l

print(L)
```

Ex 3.4.1. Explain how this code works. At what points in time do we sample the queue length?

Ex 3.4.2. The above procedure to compute the number of jobs in the system is pretty inefficient. Why is that so?

Ex 3.4.3. Try to find a(more efficient algorithm to compute L. If you cannot solve this yourself, explain my code that is provided in the hint.

3.5 MULTI-SERVER QUEUE

Let us now generalize the simulation to a queue that is served by multiple servers.

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, i.e., c times as fast as a server in the multi-server. 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 specifying the number of servers c, and change the service times as follows:

Python Code

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

3.5.2 *A real multi-server queue*

Here is the code to implement a real multi-server queue; see the queueing book to see how it works.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 4
num = 3

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

# single server queue
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)
print(W.mean(), W.std())

# code for multi server queue
c = np.array([1.0])
W = np.zeros_like(S)
w = np.zeros_like(c)
for k in range(len(S)):
    s = w.argmin()  # server with smallest waiting time
    W[k] = w[s]
    w[s] += S[k]  # assign arrival to this server
    w = np.maximum(w - X[k + 1] * c, 0)

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

Ex 3.5.2. First a test, we set the vector of server capacities `c=[1]` so that we reduce our multi-server queue to a single-server queue. Run the code and check that both simulations give the same result.

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.

Ex 3.5.3. We can modify the code for the single server queue such that it represents a single fast server working at rate c, rather than at 1.

Python Code

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

How does this work?

Now that we have tested the implementation (in part), here is the code for a queue served by three servers, all working at the same speed.

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()  # server with smallest waiting time
    W[k] = w[s]
    w[s] += S[k]  # assign arrival to this server
    w = np.maximum(w - X[k + 1] * c, 0)

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

Ex 3.5.4. Compare $\mathsf{E}[W]$ for two cases. The first is a model with 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 3.5.5. Change the code for the multi-server such that the individual servers have different speeds but such that 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 3.5.6. 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 least queueing times and variance? If the variance is affected by changing the server 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. Run this code (and write down the results). Then change the code such that the arrivals can be served on the day they arrive. Rerun and compare the results.

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).

Rerun the code with and without the arrivals being served on the period of arrival, and compare to the previous exercise. Explain the difference.

Ex 4.2.3. What are the advantages and disadvantages of using small values for T?

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 4.2.4. Which result did we use to compute $\mathsf{E}[L]$? Can we use that to estimate $\mathsf{V}[L]$?

Ex 4.2.5. Run the code and compare with the discrete time simulation.

4.2.3 *Sakasegawa's formula*

Here we use Sakasegawa's formula 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

```

```python

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

Ex 4.2.6. Explain the code, in particular the values of the parameters.

Ex 4.2.7. Run the code, and compare the value with the discrete and continuous time simulations.

Ex 4.2.8. What are advantages and disadvantages of using Sakasegawa's formula?

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, it's easy to have too large loads.

Python Code

```python
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. What does the `assert` statement? Why do I put it before doing any other work? 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

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

Ex 4.3.2. What is `W1` conceptually:?

For the queueing time, we have this:

Python Code

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

Ex 4.3.3. We set `ca2 = 1`. What is the assumption about the distribution of the interarrival times X?

Ex 4.3.4. Print `cs2`. Why is that smaller than 1? #+end$_{\text{src}}$

For the average queueing time:

Python Code

```python
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 4.3.5. In view of the previous exercise, explain that

Python Code

```python
sakasegawa(labda, ES, 1, 1)
```

gives the wrong result, but

Python Code

```python
sakasegawa(labda, ES, 1, cs2)
```

agrees with

Python Code

```python
sakasegawa(labdaB, ESB, ca2, cs2)
```

The last step:

Python Code

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

Ex 4.3.6. What is the meaning of `W3`?

The sojourn time:

Python Code

```python
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

```python
# Don't forget the copy the parameters like B so that you work with
# the same numbers.
import numpy as np

labda, B, R = 3, 13, 2
num = 30
num = B * (num // B)  # get multiple of B
X = np.ones(num + 1)
X[0] = 0
A0 = X.cumsum()
A = np.zeros_like(A0)
for i in range(num // B):
    st = i * B + 1  # start
    fi = (i + 1) * B  # finish
    print(st, fi, A0[fi])
    A[st : fi + 1] = A0[fi]
    print(A)

S0 = np.ones_like(A0)
S0[0] = 0
S = S0.copy()
S[1::B] += R
print(S)
```

Ex 4.3.7. Use the print statements to explain how the slicing, i.e., notation like `A[st :  fi +1]`, works. Explain how `A` and `S` correspond to batch arrivals and services with setup times.

Now that we know how to construct batch arrivals and job service times that include regular setups, the rest of the simulation is standard.

Python Code

```python
# put the parameters here, or glue this code after the code for
# Sakasegawa's formula

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)
```

```python
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 4.3.8. Why is `D - A` not the sojourn time?

Ex 4.3.9. 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.

4.3.4 **TODO** *Random setup times*

Maybe another year, but not this (i.e., 2022).

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. Here is the simulator.

4.4.1 *Check work*

First we should check the formulas for $\mathsf{E}[S]$ and $\mathsf{V}[S]$.

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 2 * labda
ES0 = 1 / mu
p = 1 / 20
num = 10000


S0 = np.random.exponential(scale=ES0, size=num)
R = 2 * np.ones_like(S0)
I = np.random.uniform(size=len(R)) <= p
S = S0 + R * I
print(S.mean(), S.var())

ER = R.mean()
ES = ES0 + p * ER
VS0 = ES0 * ES0
VR = R.var()
VS = VS0 + p * VR + p * (1 - p) * ER * ER
print(ES, VS)
```

Ex 4.4.1. Explain what is `I`. Then explain the line `S0 + R * I` works.

Ex 4.4.2. Run the code; include the numbers in your assignment. Are the numbers nearly the same?

4.4.2 *The simulations*

Now that we checked the formulas to compute $\mathsf{E}[S]$ and $\mathsf{V}[S]$, we can see how well Sakasegawa's formula applies for a queueing system in which a server requires regular, but random, adjustments.

Python Code

```python
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 4.4.3. Explain how D is computed.

To see how the approximation works, glue the next code below the previous code.

Python Code

```python
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


ES0 = 1 / mu
VS0 = ES0 * ES0
ER = R.mean()
ES = ES0 + p * ER
rho = labda * ES
assert rho < 1, "rho >= 1"
VR = R.var()
VS = VS0 + p * VR + p * (1 - p) * ER * ER
Cs2 = VS / ES / ES
```

```
W = sakasegawa(labda, ES, 1, Cs2, 1)
print(W)
```

Ex 4.4.4. To test the code I set at first `R = 0 * np.ones(len(A))` in the line marked as `this`. Why is this a good test?

Ex 4.4.5. 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 4.4.6. Now set `R = np.random.exponential(scale=2, size=len(A))`. What is the effect on $E[W]$? In general, do you see that indeed $E[W]$ increases with the variability of the adjustments?

Ex 4.4.7. What is the model behind this code?

Python Code

```python
import numpy as np

np.random.seed(3)

labda = 3
mu = 4
N = 1000

X = np.random.exponential(scale=1 / labda, size=N)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
S = np.random.exponential(scale=1 / mu, size=len(A))
R = np.random.uniform(0, 0.1, 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] + R[k]

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

Ex 4.4.8. In stead of

Python Code

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

we could write

Python Code

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

What modeling choice would this change reflect? Which of these two models makes the sojourn smaller?

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.

4.5.1 *Check work*

Again, first we need to check that our (implementation of the) formulas for $\mathsf{E}[S]$ and $\mathsf{V}[S]$ are correct.

Python Code

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

np.random.seed(3)

labda = 3
mu = 2 * labda
ES0 = 1 / mu
labda_f = 2
ER = 0.5
num = 10000

S0 = np.random.exponential(scale=ES0, size=num + 1)
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()

A = 1 / (1 + labda_f * ER)
ES = ES0 / A
print(ES, S.mean(), S0.mean() + N.mean() * R.mean())

C02 = 1
Cs2 = C02 + 2 * A * (1 - A) * ER / ES
print(Cs2, S.var() / (S.mean() ** 2))
```

Ex 4.5.1. Run this code, and check the result. Then chance `num` to 100000 to see that the estimate improves.

Ex 4.5.2. Explain how we compute `S[i]`.

4.5.2 *The simulations*

Here is all the code to compare the results of a simulation to Sakasegawa's formula.

Python Code

```python
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

S0 = np.random.exponential(scale=ES0, size=num + 1)
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()


X = np.random.exponential(scale=1 / labda, size=num)
A = np.zeros(len(X) + 1)
A[1:] = X.cumsum()
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())


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


A = 1 / (1 + labda_f * ER)
ES = ES0 / A
C02 = 1
Cs2 = C02 + 2 * A * (1 - A) * ER / ES
rho = labda * ES
assert rho < 1, "rho >= 1"
W = sakasegawa(labda, ES, 1, Cs2, 1)
print(W)
```

Ex 4.5.3. Run the code and include the results in your assignment.

Ex 4.5.4. 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 $\mathsf{E}[W]$? (Use Sakasegawa's formula for this; you don't have to do the simulations. In general, computing functions is much faster than simulation.)

4.6 A SIMPLE TANDEM NETWORK

We have two queues in tandem. Again we compare the approximations with simulation.

4.6.1 *Simulation in continuous time*

This code simulates two queues in tandem in continuous time.

Python Code

```python
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. Explain how the code works. Why did I choose these parameters?

4.6.2 *Sakasegawa's formula plus tandem formula*

Now we can check the quality of the approximations.

Python Code

```python
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
mu1 = 1.2 * labda
ES1 = 1 / mu1
rho1 = labda * ES1
Ca2 = 1
Cs2 = 1
W1 = sakasegawa(labda, ES1, Ca2, Cs2, 1)
Cd2 = rho1 ** 2 * Ca2 + (1 - rho1 ** 2) * Cs2

Ca2 = Cd2
mu2 = 1.1 * labda
ES2 = 1 / mu2
Cs2 = 1
W2 = sakasegawa(labda, ES2, Ca2, Cs2, 1)


J = W1 + ES1 + W2 + ES2
print(W1, ES1, W2, ES2, J)
```

Ex 4.6.2. How does the code work? Compare the results obtained from simulation and by formula.

Ex 4.6.3. Assume that the service times at the first queue are constant and equal to $1/\mu_1$. What should be the parameter values for Sakasegawa's formula (explain). Then run both codes and comment on the results.

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

```python
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 and check that indeed long waiting times do occur. What does the `this` line do?

Ex 5.2.2. Give the Kendall notation for the queueing model that we simulate here.

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 realize this, change the for loop to

Python Code
```python
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 5.2.3. Run this code and check its effect on $\mathsf{E}[W]$, $\mathsf{V}[W]$, and $\max\{W\}$. Explain how it can occur that the maximum can still be larger than T.

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.

However, modifying this code so that I can block seems not straightforward.

Ex 5.2.4. The recursion we use, i.e.,

Python Code
```python
for k in range(1, len(S)):
    W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)
```

assumes that job $k-1$ is accepted. Suppose that `W[k] >= T` so that job k is blocked. Explain that with this recursion, we now have a problem to compute the waiting time for job $k+1$.

Here is better code.

Python Code
```python
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 5.2.5. What does the vector `I` represent?

5.2.4 *Some other blocking rules*

There are other rules to block jobs.

Ex 5.2.6. In this 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] + S[k] >= T:
        I[k] = 0

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

how does the rule below block jobs?

Ex 5.2.7. Likewise,

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())
```

how does the rule below block jobs? What is the meaning of `V`?

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)

```

```python
num = 100000
labda = 1
mu = 1.1 * 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)

print(S.mean(), W.mean(), W.max())
rho = labda / mu
print(rho**2 / (1 - rho))
```

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? Why should `W.mean()` and $\rho^2/(1-\rho)$ be approximately equal

5.3.2 *Include blocking*

Here is the code with a blocking rule.

Python Code

```python
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. Explain how the code works. What do the printed KPIs denote? Finally, why is the `this` line very dangerous to use, hence to be avoided? (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 $\{L\}$ are easy, while in continuous time the recursions for $\{W\}$ or $\{J\}$ 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

```python
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 there no problem with the statement `(L == 0)`?

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. 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

```python
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. At first I had these lines:

Python Code

```python
elif len(Q) < T:
    Q.append(k)
    D[k] = D[Q[-1]] + S[k]
```

Why is that wrong?

Ex 5.4.7. What goes wrong with this code:

Python Code
```python
if len(Q) < T:
    D[k] = max(D[Q[-1]], A[k]) + S[k]
    Q.append(k)
```

Ex 5.4.8. What queueing discipline would result if we would use the `pop()` and `appendleft()` methods of a `deque`?

Ex 5.4.9. What queueing discipline would result if we would use the `pop()` and `append()` methods of a `deque`?

Ex 5.4.10. 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
```python
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 5.4.11. 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 $M/M/1$ queue?

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

5.5 AN ALGORITHM FOR THE $M/G/1$ QUEUE WITH BLOCKING

In the queueing book we develop an algoritm to compute $\pi(n)$. Here we implement this, use this as another test on the simulator, and improve our understanding of queueing systems.

5.5.1 *The algorithm*

This is the code.

Python Code
```python
import numpy as np
from scipy.integrate import quad
from scipy.stats import expon

np.random.seed(3)
```

```

labda = 1
mu = 1.2 * labda
T = 5
S = expon(scale=1 / mu)


def g(j, x):
    res = np.exp(-labda * x) * (labda * x) ** j * S.pdf(x)
    return res / np.math.factorial(j)


f = np.zeros(T + 1)
for j in range(T + 1):
    f[j] = quad(lambda x: g(j, x), 0, np.inf)[0]

F = f.cumsum()
G = 1 - F

pi = np.ones(T + 1)
for n in range(T):
    pi[n + 1] = pi[0] * G[n]
    pi[n + 1] += sum(pi[m] * G[n + 1 - m] for m in range(1, n + 1))
    pi[n + 1] /= f[0]

pi /= pi.sum()
print(pi)
```

Ex 5.5.1. Which formulas (give the numbers) of the queueing book have we implemented?

Ex 5.5.2. Run this code after the computation of `f`.

Python Code

```
j = 2
print(mu / (mu + labda) * (labda / (labda + mu)) ** j, f[j])
```

Why should these numbers be the same?

Ex 5.5.3. Run the code for $\mu = 0.3$ and compare the numerical results to what you get from:

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()
print(pi)
```

Explain why you should get the same numbers.

Ex 5.5.4. When the service times are contant, explain that this code computes `f` correctly:

Python Code
```python
from scipy.stats import expon, uniform, poisson
# include useful code here
f = poisson(labda / mu).pmf(range(T + 1))
```

Then change the `S` in the simulation part to

Python Code
```python
S = np.ones(len(A)) / mu
```

Run the code and include your results; of course the simulation and the algoritm should give more or less the same results.

Ex 5.5.5. As another good example, take $S \sim U(0, 2/\mu)$. The relevant code changes are this:

Python Code
```python
from scipy.stats import expon, uniform, poisson
# other stuff for the model
S = uniform(0, 2 / mu)
```

and for the simulator:

Python Code
```python
S = np.random.uniform(0, 2 / mu, size=len(A))
```

Run the code, and include your output.

5.5.2 *Effect of blocking on performance*

Ex 5.5.6. Take $\lambda = 1$ and $\mu = 1.1$. Use the algorithm to compute the loss probability and $\mathsf{E}[L]$ for $T = 5$, $T = 10$ and $T = 15$. Include the numbers.

Ex 5.5.7. Do the same computations for $\mu = 0.5\lambda$. Why is the loss probability not so sensitive to T?

Ex 5.5.8. Set $\mu = 1.2\lambda$ again. Then compare the loss probability for $T = 5, 10, 15$ for $S \sim \text{Exp}(\mu)$ and $S \sim U(0, 2/\mu)$. What is the influence of service time variability on the loss when $T = 5$, $T = 10$, $T = 15$? Why is this influence relatively more important for larger T?

5.6 SERVER CONTROL

With blocking we control whether jobs are allowed to enter the system. We can also focus on another type of control, namely that of the server. Here we show how to simulate a system in which the server switches on when the waiting time exceeds a level D. When the server is empy again, it switches off.

Let us first plot the virtual waiting time. Earlier we discussed how to plot the virtual waiting time for a given array of waiting times, arrival times and departure times. Thus, the only relevant code is how to find the waiting time under a D policy.

Note that we here use the letter T to refer to the threshold since the letter D is already given to the departure times.

Python Code

```python
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(S)
On = False
for k in range(1, len(S)):
    if On:
        W[k] = max(W[k - 1] + S[k - 1] - X[k], 0)
        On = False if W[k] <= 0 else True
    else:
        W[k] = W[k - 1] + S[k - 1]
        On = False if W[k] < T else True
    D[k] = A[k] + W[k] + S[k]

idx = np.where(W <= 0)[0]

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/D-policy.pdf")
```

Ex 5.6.1. Explain how the waiting times are computed.

Given cost K to switch on the server and holding h (per unit waiting time per unit time) we want to find the threshold T that minimizes the time-average cost. The code below shows how to compute the cost for a given T.

Python Code

```
h = 1.0
K = 3
cost = 0
epoch, height = E[:, 0], E[:, 1]
for i in range(1, len(epoch)):
    dx = epoch[i] - epoch[i - 1]
    dy = (height[i] + height[i - 1]) / 2.0
    cost += h * dx * dy
    if dy == 0:
        cost += K
print(cost / D[-1])
```

Ex 5.6.2. Explain how the code works. What is `dx`, what is `dy`? Why do we divide by 2?

Ex 5.6.3. Explain the procedure to find the best T.

6 QUEUEING THEORY ASSIGNMENT: SIMULATION WITH EVENT STACKS

6.1 GENERAL INFO

This file contains the code and the results that go with this youtube movie:

We will discuss *heap queues* and *classes*. With these we will build a very powerful simulation environment. You should memorize in particular that we implement an *event stack* with a heap queue.

6.1.1 ***TODO*** *Set theme and font size*

Set the theme and font size so that it is easier to read on youbute

```
(load-theme 'material-light t)
(set-face-attribute 'default nil :height 240)
```

By the way, this is emacs lisp; you cannot run it in python.

6.1.2 *Load standard modules*

I need a new modules `heapq`.

Python Code

```python
from heapq import heappop, heappush
import numpy as np
from scipy.stats import expon, uniform

import matplotlib.pylab as plt
import seaborn as sns; sns.set()

np.random.seed(3)
```

6.2 SORTING WITH HEAP QUEUES

Here is a simple example of using heaps to sort data

We are going to sort a bunch of students by age. The heap uses the first element of the *tuple* of a student's age and name to sort the students.

Ex 6.2.1. Look up on the web: what is the difference between a tuple and a list? When to use one or the other? (As always, keep your answer brief.)

Let's put a number of students on the stack.

Python Code

```python
stack = []

heappush(stack, (25, "Cynthia"))
heappush(stack, (21, "James"))
heappush(stack, (21, "James"))
heappush(stack, (25, "Cynthia"))
```

```python

print(stack)
```

Python Code
```python
age, name = heappop(stack)
print(stack)
```

Python Code
```python
age, name = heappop(stack)
print(stack)
```

Python Code
```python
heappush(stack, (20, "Pete"))
heappush(stack, (18, "Clair"))
heappush(stack, (14, "Jim"))

print(stack)
```

With popping, we take an entry from the top of the stack.

Python Code
```python
age, name = heappop(stack)
print(stack)
```

Python Code
```python
heappush(stack, (15, "John"))
print(stack)
```

Ex 6.2.2. Add an extra attribute to the tuple, for instant a student's height, and make a stack by which you can sort by height. Include your code.

You must have seen during the youtube video that by putting the same element more than once on the stack resulted in a longer, and unsorted, stack. (This came to a surprise to me!) Here I made a simple mistake. A heap queue is not necessarily sorted. However, when *popping* from the stack, it is guaranteed that the elements are popped in sorted order. In other words, it is irrelevant how the stack is sorted, as long as the elements come out in a sorted way. Let's check this.

Python Code
```python
stack = []

heappush(stack, (25, "Cynthia"))
heappush(stack, (21, "James"))
heappush(stack, (21, "James"))
heappush(stack, (25, "Cynthia"))

print(stack)
```

Apparently, the stack does not appear to be sorted. In the code below I pop the elements of the stack, with `heappop`, and append the popped items to the list `res`. You should know that anytime we pop from the stack, the stack becomes one element shorter. So the code below moves elements from stack to res. Any pass through the while loop removes an element of the stack, and the loop keeps running until the stack is empty. (To help you think about this, this is an ideal exam question.)

Python Code
```python
res = []

while stack:
    res.append(heappop(stack))

print(res)
```

Indeed, res *is* sorted.

6.3 CLASSES

In a class we organize information.

Python Code
```python
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. After you have read this section on classes, extend the class such that we can give the student also a surname. Include your code.

Ex 6.3.2. Look up on the web: what does the repr method do?

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

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

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

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

Ex 6.3.3. Make two more students, e.g., take your phones, ages and names. (And if you don't like to spread such details, just lie about your age :-)) Then add these students to the stack. Show your code and the output.

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

Python Code
```
stack = []

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

res = []  # get a sorted list  students in order of their name.
while stack:
    res.append(heappop(stack))

print(res)

```

Note once more that we can provide a *key* as a first element in a tuple, and then insert the entire tuple into a heap. In the rest of the tuple we can put anything we like. Like this, we associate things (here student objects) to keys.

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. Then sort by name.

6.4 SORTING JOBS WITH A HEAP QUEUE

How can use heap queues in the simulation of queueing systems? To see, think of time as a sequence of events in which things happen.Then, in a queueing system, two things can happen: a job can arrive or a job leaves. So, by specifying the arrival times of jobs, and their departure times, and storing these times in a heap queue, we use the heap queue to sort all these times. Then we jump from event to event, and this is nothing but the queueing simulation!. In other words, with a heap queue, we can let the heap queue do all the work of tracking time. You'll see below we can nearly forget about it. Once you get the idea (which takes some time), you'll see how neat is all is.

6.4.1 *A decent job class*

We store the arrival, service, and departure time as job *attributes*. We also store the queue length at arrival times to gather statistics at the end. We can then compute the job sojourn time by means of *class methods*.

Let's start from scratch again with the code so that you have a simple starting point.

Python Code
```
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):
            return self.sojourn_time() - self.service_time

        def service_start(self):
            return self.departure_time - self.service_time

        def __repr__(self):
            return f"{self.arrival_time}, {self.service_time}, {self.service_start()}, \
                    {self.departure_time}\n"

        def __le__(self, other):
            # this is necessary to sort jobs when they have the same arrival times.
            return self.id <= other.id
```

Ex 6.4.1. Explain the waiting time and sojourn functions.

Let's make a few jobs and store them in a heap queue. For later purposes, we have to add labels to indicate what type of event we are dealing with, an arrival or a departure.

Python Code

```
ARRIVAL, DEPARTURE = 0, 1

events = []  # event stack, global
num_jobs = 5

time = 0
for i in range(num_jobs):
    time += 3
    job = Job()
    job.arrival_time = time
    job.service_time = 5
    heappush(events, (job.arrival_time, job, ARRIVAL))

while events:
    time, job, typ = heappop(events)
    print(job)
```

6.5 A VERY POWERFUL GG1 SERVER SIMULATOR

6.5.1 *The class definition*

We need two states to indicate whether the server is busy or idle.

Python Code

```
IDLE, BUSY = 0, 1
```

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
```

```python
        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 handle departure method works.

Ex 6.5.2. Explain how the run method works.

6.5.2 *Generating jobs*

Python Code

```python
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()
```

Ex 6.5.3. Suppose we would write

Python Code

```
gg1 = GG1(G, F, num_jobs)
```

What are then the arrival rate and service rate?

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 the following.

Python Code

```
from itertools import islice

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

Ex 6.5.4. Read on the web what `islice` does, and explain it in your own words.

Ex 6.5.5. 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 6.5.6. Explain the contents of the above table; what do the rows represent? Why is the content of column 5 negative?

6.5.3 *Run the simulation*

Now run the simulation.

Python Code

```
gg1.run()
```

Print the first 5 jobs (not all, because when we simulate a 1000 or so jobs, the document explodes).

Python Code

```
print(gg1.served_jobs[:5])
```

Ex 6.5.7. Explain the first two lines. Are the results in line with what you expect how a $G/G/1$ FIFO queue should behave? If so, why (not)?

6.5.4 *Analyze the results, statistics*

Here is a plot.

Python Code

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

Ex 6.5.8. 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.

And some statistics. For instance the sojourn times.

Python Code

```python
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())
```

Python Code

```python
wait = sojourn.mean() - G.mean()
print(wait)
```

6.5.5 *Comparison with Sakasegawa's formula*

We can compare the results of our simulation with the theoretical values of the stationary state.

Python Code

```python
def sakasegawa(F, G, c):
    labda = 1.0 / F.mean()
    ES = G.mean()
    rho = labda * ES / c
    EWQ_1 = rho ** (np.sqrt(2 * (c + 1)) - 1) / (c * (1 - rho)) * ES
    ca2 = F.var() * labda * labda
    ce2 = G.var() / ES / ES
    return (ca2 + ce2) / 2 * EWQ_1


average_wait = sakasegawa(F, G, c=1)
print(average_wait, average_wait + G.mean())
```

Ex 6.5.9. Compare the result of Sakasegawa's formula and the results of your simulation (with about 100 jobs). You should remark that the difference in the waiting time is quite large. Do we simulate the $M/M/1$ queue here? Is Sakasegawa's result is exact for the $M/M/1$ queue?

Ex 6.5.10. Run an example with more jobs, 1000 or so. Is now the average sojourn time in the simulation nearer to the theoretical result?

Ex 6.5.11. Another idea is this. We start with an empty system, and that is a very particular state. Suppose we would start with 10 jobs in queue. Do a simulation for 100 jobs, but with 10 jobs in queue at the start. What is then the average sojourn time? Hint, below I include some code to put 10 jobs in queue right at the start. There is subtlety. Af first I set the arrival time to 0 for all these jobs, but then the heapq algorithm doensn't know how to sort the jobs. To repair this problem, I just take very small arrival times.

Python Code

```
eps = 0.0001
for i in range(10):
    job = Job()
    job.arrival_time = i*eps
    job.service_time = G.rvs()
    heappush(events, (job.arrival_time, job, ARRIVAL))
```

6.5.6 *Further experiments*

Ex 6.5.12. Change the service distribution to the uniform distribution $U[a,b]$. Take a and b reasonable, so that the load remains below 1. Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times.

Ex 6.5.13. Change the arrival distribution to a uniform distribution, and take also the service distribution to be uniformly distributed. Take some sensitble values for λ and $\mathsf{E}[S]$, e.g., $\lambda = 1$ and $\mathsf{E}[S] = 0.4$. Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times.

6.6 SPTF SCHEDULING

Suppose we prefer to serve the shortest job in queue; it can be proven that this scheduling rule minimizes the queue length. We can inherit all of our `GG1` queue, except the scheduling rule. For this we need to change just one line!

Python Code

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

And that's it! Now run it.

Python Code

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

6.7 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 6.7.1. Run an example with 100 jobs. Make a graph of the waiting times and the sojourn times. Comment on your findings.

6.8 serve longest job first

Ex 6.8.1. Update the code of the SPTF queue such that the longest job is selected from the queue, rather than the shortest. (Hint, put a minus sign at the right position when adding a job to the queue heap.) Simulate it. Make a graph of the waiting times and the sojourn times. Comment on your findings.

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. Observe that for D_1 we need D_0. If A_0 would be the arrival time of the first job, then what would we take for D_{-1}?

h.3.2.8. We subtract `S[0]` as if we served the corresponding job, but did we actually serve it?

h.3.3.7. 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.3.4.3. Here is the code.

Python Code

```python
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)
```

```python
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")
```

h.3.5.5. 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.4.2.4. Recall $L = \lambda W$.

h.4.3.4. When $S = R/B + S_0$, then part of S is constant. Hence, can it be (relatively) as variable as S_0?

h.4.3.7. Why do I chose `np.ones` to fill `A0` and `S0`? What is the difference between `A0` and `A`, and `S0` and `S`?

h.4.3.8. What do you think I used first?

h.4.4.8. What is the influence on the setup? Do we still require that the setup has to be done immediately before a service starts?

h.4.5.4. For instance, set `labda_f = 4` and `ER = 0.25`.

h.4.6.1. For what situations are the formulas exact?

h.4.6.3. In the simulation, replace the line with `S1` by `S1 = np.ones(len(A1)) / mu1`. In the formula's, don't forget to update C_s^2 for the relevant queue.

h.5.2.2. It's not the $D/D/1$ queue.

h.5.2.3. Check the section in the queueing book on the $M^X/M/1$ queue, in particular the section on blocking. Which blocking method do we include here?

h.5.2.4. We want to block job k when $W_k \geq T$. That means that job $k+1$ should not see S_k, but (as far as I see) we cannot convert that into an elegant recursion for W_{k+1}

h.5.2.5. If `I[k] == 1`, then what happens to job k?

h.5.3.2. Why is this the $M/M/1$ queue when the batches `B = np.random.randint(1, 2, size=num)`? (Recall my obsession with testing code.)

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. To what job does `Q[-1]` point to? It's one to far!

h.5.4.7. When `Q` is empty (i.e., `len(Q)` is 0), then what is `Q[-1]`?

h.5.4.8. Does it matter whether we push jobs from right to left through a quue, rather then from left to right?

h.5.4.9. It's not FIFO.

h.5.4.10. Is `L.max()` larger than 100 for this simulation?

h.5.5.6. Why is the loss probality equal to π_T?