Queueing simulations

N.D. Van Foreest and E.R. Van Beesten

January 23, 2019

Contents

1 Introduction

Typically a queueing system is subject to rules about when to allow jobs to enter the system or to adapt the service capacity. Such a decision rule is called a *policy*. The theoretical analysis of the efficacy of policies is often very hard, while with simulation it becomes doable. In this document we present a number of cases to see how simulation can be used to analyze and improve queueing systems. Besides the fact that these the cases will improve your understanding of queueing systems and probability theory, they will also make clear that simulation is a really creative activity and involves solving many interesting and challenging algorithmic problems.

Each case is organized in a number of exercises. For each exercise,

1. Make a design of how you want to solve the problem. For instance, make a model of a queueing system, or a control policy structure, or compute relevant KPIs (key performance indicators, such as cost, or utilization of the server, and so on). In other words, think before you type.

2. Try to translate your ideas into pseudo code or, better yet, python[1]

3. If you don't succeed in getting your program to work, look up the code written by us and type it into your python environment.[2].

4. When an exercise has a hint, its marked in the margin with a penguin symbol[3].

5. Simulate a number of scenarios by varying parameter settings and see what happens.

We expect you to work in a groups of 2 to 3 students and bring a laptop with an *installed and working* python environment, preferably the anaconda package available at https://www.anaconda.com/, as this contains all functionality we will need[4]. You can find a nice tutorial to python programming at https://www.programiz.com/python-programming. Note, this site advises to install python just by itself. We

---

[1]Some of you might wonder why we use python rather than R. There are a few reasons for this. Python is more or less the third most used programming language, after C++ and java. It is widely used by companies, while R is hardly used outside academia. Programming OR applications is easier in python; it will also used in other courses. Finally, if you are interested in machine learning and artificial intelligence, python i, hands-down the best choice.

[2]Typing yourself forces you to read the code well.

[3]A penguin is the logo for Linux. As using Linux will make you a happier person overall, it seems fitting to use the Linux logo as the messenger of good news.

[4]There are also python environments available on the web, such as repl.it., but that is typically a bit less practical than running the code on your own machine.

instead advise you to download anaconda, as this contains also the required numerical libraries.

We will use the following libraries of python a lot:

- `numpy` provides an enormous amount of functions to handle large (multi-dimensional) arrays with numbers.

- `scipy` contains numerical recipes, such as solvers for optimization software, solvers for differential equations. `scipy.stats` contains many probability distributions and numerical methods to operate on these functions.

- `matplotlib` provides plotting functionality.

Our code is not the most efficient, or fast. Rather, we focus on clarity of code so that the underlying reasoning is as clear as possible. Once our ideas and code are correct, we can start optimizing, if this is necessary.

Finally, the code is part of the course, hence of the midterms and the exams. Unless indicated as not obligatory, you have to be able to read the code and understand it.

2 Exponential distribution

The aim of this tutorial is to show, empirically, a fascinating fact: even for very small populations in which individuals decide independently to visit a server (a shop, a hospital, etc), the exponential distribution is a good model for the inter-arrival times as seen by the server. We will develop a simulation to motivate this 'fact of nature'. In particular, our aim is to build analogues to Figure 1.1, 1.2., and 1.3 of the queueing book[5], but in terms of cdfs instead of pdfs. (Read the description that underlies these figures.)

Exercise 2.1. Make a plan of the steps you have to carry out to make Figure 1.1 of the queueing book. In the next set of exercises we'll carry out these steps. So please do not read on before having thought about this problem, but spend some 5 minutes to think about how to approach the problem and how to chop it up into simple steps. Then organize the steps into a logical sequence. Don't worry at first about how to convert your ideas into computer code. Coding is a separate activity. (As a matter of fact, I always start with making a small plan on how to turn an idea into code, and I call this step 'modeling'. Typically this is a creative step, and not easy.)

We need some python libraries to make our lives a bit easier. You should copy this code into your editor.

```python
import numpy as np
import scipy
import matplotlib
matplotlib.use('pdf')
import matplotlib.pyplot as plt

# this is to print not too many digits to the screen
np.set_printoptions(precision=3)
```

2.1 Empirical distributions

One important step in this process is to compute the empirical distribution. As this is much more interesting (and challenging) than you might think[6], we start with this. Once we can compute empirical distribution functions, we are in good shape to set up the rest of the simulation.

---

[5]You can find this here: https://github.com/ndvanforeest/queueing_book

[6]If you search the web, you will see that computing the empirical density function is even more challenging.

Before designing an algorithm to compute , it is best to start with a simple numerical example and try to formalize the steps we take in the process.

Exercise 2.2. Suppose you are given the following numbers:

```
a = [3.2, 4, 4, 1.3, 8.5, 9]
```

What steps do you take to make the empirical distribution function? Recall, this is defined as

$$F(x) = \frac{\#\{i : a_i \leq x\}}{n}, \tag{1}$$

with n is the length of a.

Can you turn it into an algorithm? (Just attempt to design an algorithm. Even if you don't succeed, trying is important. Then read the code in the solution.)

Exercise 2.3. The method provided by the (solution of the) previous exercise is simple, but not completely correct. What is wrong?

A better idea is to consider the sorted version s of a.

Exercise 2.4. Sort the numbers in the list a; let this be s. Make a plot (by hand) of s. Now observe the important fact that $i \rightarrow s_i$ is the inverse of the distribution F (except for the normalization).

Exercise 2.5. Find now a way to invert $i \rightarrow s_i$, normalize the function to get a distribution, and make a new plot.

Exercise 2.6. In the previous exercise (read the solution), we start y with 1 and end with `len(a)+1`. Why is that?

You should know that for loops in python are quite slow (and for loops in R seem to be really dramatic). For large amounts of data it is better to use `numpy`.

Exercise 2.7. Use the `numpy` functions `arange`, to replace the `range`, and `sort` to speed up the algorithm of the previous exercise.

With the algorithm of Exercise 2.7 we can compute and plot a distribution function of inter-arrival times specified by a list (vector, array) a. For our present goals this suffices. If, however, you like details, you should notice that our plot of the distribution function is still not entirely OK: the graph should make jumps, but it doesn't. Moreover, our cdf is not a real function, it can be of the form $x = (1,1,3)$, $y = (0,0.5,1)$. In the rest of this subsection we repair these points. You can skip this if you are not interested.

Exercise 2.8. Read about the `drawstyle` option of the `plot` function of `matplotlib` to see how to make jumps.

Exercise 2.9. Finally, we can make the computation of the cdf significantly faster with using the following `numpy` functions.

1. `numpy.unique`

2. `numpy.sort`

3. `numpy.cumsum`

4. `numpy.sum`

How can you use these to compute the cdf?

2.2 Simulating the arrival process of a single customer

The next step is to simulate inter-arrival times of a single customer and make an empirical cdf of these times. Then we graphically compare this cdf to the exponential distribution. A more formal method to compare cdfs is by means of the Kolmogorov-Smirnov statistic (see Wikipedia) which we develop in passing.

Exercise 2.10. Generate 3 random numbers uniformly distributed on [4,6]. Print these numbers to see if you get something decent. Read the documentation of the `uniform` class of in the `scipy.stats` module[7]. Check in particular the `rvs()` function.

Exercise 2.11. Generate $L = 300$ random numbers $\sim U[4,6]$ and make a histogram of these numbers. You should interpret these random numbers as inter-arrival times of one customer at a shop (in hours say.)

Exercise 2.12. Compute the empirical distribution function of these inter-arrival times, and plot the cdf.

Exercise 2.13. We would like to numerically compare the empirical distribution of the inter-arrival times to the theoretical distribution, which is uniform in this case. For this we can use the Kolmogorov-Smirnov statistic. Try to come up with a method to compute this statistic, and then compute it.

You might find the following functions helpful (read the documentation on the web to see what they do):

---

[7]When coding you should develop the habit to look up things on the web

1. `numpy.max`

2. `numpy.abs`

3. `scipy.stats.uniform`, the `cdf` function.

Exercise 2.14. Now compute the KS statistics to compare the simulated inter-arrival times with an exponential distribution with a suitable mean. (What is this suitable mean?).
See `scipy.stats.expon`.

Exercise 2.15. Finally, plot the empirical distribution and the exponential distribution in one graph. Explain why these graphs are different.

2.3 Simulating many customers

We would now like to simulate the inter-arrival process as seen by a shop that serves many customers. For ease, we call this the merged, or superposed, inter-arrival process. Again, this requires quite a bit of thought. Thus, we start with a numerical example with two customers, and organize all steps we make to compute the empirical distribution of the merged inter-arrival process. Then we make an algorithm, and scale up to many numbers.

Exercise 2.16. Suppose we have two customers with inter-arrival times $a = [4, 3, 1.2, 5]$ and $b = [2, 0.5, 9]$. Make, by hand, the empirical cdf of the merged process.

Exercise 2.17. The steps of the previous exercise can be summarized by this code:

```python
from itertools import chain

def superposition(a):
    A = np.cumsum(a, axis=1)
    A = list(sorted(chain.from_iterable(A)))
    return np.diff(A)
```

Try to understand this by reading the documentation (on the web) of the following functions.

1. `numpy.cumsum`, in particular read about the meaning of axis.

2. `itertools.chain.from_iterable`

3. `numpy.diff`

Exercise 2.18. Generate 100 random inter-arrival times for 3 customers, plot the cdf of the merged process, and compare to the exponential distribution with the correct mean. Also compute the Kolmogorov-Smirnov statistic for this case. What is the effect of increasing the number of customers from $N = 1$ to $N = 3$?

Exercise 2.19. Compare the empirical distribution of the inter-arrival times generated by $N = 10$ customers to the exponential distribution (compute the appropriate arrival rate). Make a plot, and explain what you see.

Exercise 2.20. Do the same for $N = 10$ customers with normally distributed inter-arrival times with $\mu = 5$, and $\sigma = 1$. For this use `scipy.stats.norm`. What do you see? What is the influence of the distribution of inter-arrival times of an individual customer?

Exercise 2.21. If $\mu = \sigma = 5$ then the analysis should break down. What happens if you set $\sigma = 5$? If you don't get real strange results, the code itself must be wrong[8].

2.4 Summary

Exercise 2.22. Make a summary of what you have learned from this tutorial.

---

[8]When testing code it is also a good idea to see what happens if you use bogus numbers. The program should fail or give very strange results.

3 Simulation of the *G/G*/1 queue in discrete time

In this tutorial we simulate the queueing behavior of a supermarket or hospital, in fact, for a general service system. We first make a simple model of a queueing system, and then extend this to cover more and more difficult queueing situations. With these models we can provide insight into how to design or improve real-world queueing systems. You will see, hopefully, how astonishingly easy it is with simulation to evaluate many types of decisions and design problems.

For ease we consider a queueing system in discrete time, and don't make a distinction between the number of jobs in the system and the number of jobs in queue. Thus, queue length corresponds here to all jobs in the system, c.f. Section 1.4 of the queueing book.

3.1 Set up

Exercise 3.1. Write down the recursions to compute the queue length at the end of a period based on the number of arrivals a_i during period i, the queue length Q_i at the start, and the number of services s_i. Assume that service is provided at the start of the period.

Use a for-loop to sketch an algorithm (in pseudo code). Then check the solution for the python code.

With the code of the above exercises we can start our experiments.

Exercise 3.2. Copy the code of the previous exercise to a new file in Anaconda. Then add the code below and run it. Here λ is the arrival rate, μ the service rate, N the number of periods, and q_0 the starting level of the queue. Explain what the code does. Can you also explain the value of the mean and the standard deviation?

```
labda, mu, q0, N = 5, 6, 0, 100

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)
print(a.mean(), a.std())
```

Exercise 3.3. Modify the appropriate parts of code of the previous exercise to the below, and run it. Explain what you see.

```
labda, mu, q0, N = 5, 6, 0, 100
a = poisson(labda).rvs(N)
```

```
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0)

plt.plot(Q)
plt.show()
```

Exercise 3.4. Getting statistics is now really easy. For example, sketch the empirical distribution function of the queue length process and Compute the mean number of departures per period.

Exercise 3.5. Can you explain the value of the mean number of departures?

Exercise 3.6. Plot the queue length process for a large initial queue, for instance, with

```
q0, N = 1000, 100

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0)

plt.plot(Q)
plt.show()
```

Next, set $q_0 = 10000$ and $N = 1000$. (In Anaconda you can just change the numbers and run the code again, in other words, you don't have to copy all the code.) Finally, make these values again 10 times larger.

Explain what you see. What is the drain rate of Q?

Exercise 3.7. What do you expect to see when $\lambda = 6$ and $\mu = 5$? Once you formulated your hypothesis, check it.

```
N = 10000
labda = 6
mu = 5
q0 = 0

a = poisson(labda).rvs(N)
```

```
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0)

plt.plot(Q)
plt.show()
```

Explain again what you see.

3.2 What if analysis

Hopefully you are convinced by now about how powerful simulation is. We can start asking all kinds of questions.

Exercise 3.8. For instance, the mean and the sigma of the queue length might be too large, that is, customers complain about long waiting times. Suppose we are able, with significant technological investment, to make the service time more predictable. What would be the influence of this?

To quantify the effect of regularity of service times we first assume that the service times are exponentially distributed, then we change it to deterministic times. Deterministic service times are the best we can achieve. Thus, if we don't change the mean of the service times, it will not become any better than this.

So, let's test. First run this:

```
N = 10000
lbw = 6
mu = 5
q0 = 0

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0)
print(Q.mean(), Q.std())
```

Then run the same code with `s = np.ones_like(a) * mu` rather than the Poisson distribution. Explain the result.

Exercise 3.9. Next, we might be able to reduce the average service time by 10%, say, but reducing the variability is hard. To see the effect of this change, replace s by

```
s = poisson(1.1*mu).rvs(N)
```

i.e., we increases the service rate with 10%.

Do the computations and compare the results to those of the previous exercise. What change in average service time do we need to get about the same average queue length as the one for the queueing system with deterministic service times? Is an 10% increase enough, or should it be 20%, or 30%? Just test a few numbers and see what you get.

You should make the crucial observation now that we can experiment with all kinds of changes (system improvements) and compare their effects. In more general terms: simulation allows us to do 'what-if' analysis.

3.3 Control

In the previous section we analyzed the effect of the design of the system, such as changing the average service times. These changes are independent of the queue length. In many systems, however, the service rate depends on the dynamics of the queue process. When the queue is large, service rates increase, while when the queue is small, service rates decrease. You see this type of policy often in supermarkets, extra cashiers will open when the queues increase.

Exercise 3.10. Suppose that normally we have 6 servers, each working at rate 1 per period. When the queue becomes longer than 20 we hire two extra servers, and when the queue is empty again, we send the extra two servers home, until the queue hits 20 again, and so on. Try to write your own code to compute the queue process. (This is pretty challenging, and a good test to see how creative you are in modeling.)

Exercise 3.11. Another way to deal with large queues is to simply block customers if the queue is too long. What if we block at a level of 15? How would that affect the average queue and the distribution of the queue?
Modify the code to compute the queue and do an experiment.

As a final case consider a single server queue that can be switched on and off. There is a cost h associated with keeping a job waiting for one period, there is a cost p to hire the server for one period, and it costs s to switch on the server. Given the parameter values of the next exercise, would what be a good threshold N such that when the queue hits or exceeds N, the server is switched on? We assume (and it is easy to prove) that it is optimal to switch off the server when the queue is empty.

Exercise 3.12. Jobs arrive at rate $\lambda = 0.3$ per period, if the server is present the service rate is $\mu = 1$ per period. The number of arrivals and service are Poisson

distributed with the given rates. Then, $h = 1$ (without loss of generality), $p = 5$ and $S = 500$. Write a simulator to compute the average cost for setting $N = 100$. Then, change N to find a better value.

Exercise 3.13. What have you learned from this tutorial? What interesting extensions, relevant for practice, can you thing of?

4 Simulation of the $G/G/1$ queue in continuous time

In this tutorial we will write a simulator for the $G/G/1$ queue. For this we use two concepts that are essential to simulate any stochastic system of reasonable complexity: an *event stack* (or event schedule) to keep track of the sequence in which events occur, and *classes* to organize data and behavior into single logical units. We will work in steps towards our goal; along the way you will learn a number of fundamental and highly interesting concepts such as *classes* and efficient *data structures*. If you have understood the implementation of the $G/G/1$ queue, discrete event simulators are no longer a black box for you. Hence, what you will learn from this practical extends well beyond the simulation of the $G/G/1$ queueing process. Once we have built our simulator we apply it to a case in which we analyze the queueing behavior at an check-in desk at an airport with a single server.

In Section 5 we will generalize the simulator to the $G/G/c$ queue, clean up the code by working with a class for the $G/G/c$ queue. We can then also extend the case accordingly.

4.1 Sorting with event stacks

If we simulate a stochastic system we like to move from event to event. To illustrate this idea, consider a queueing process such as the $M/M/1$ queue. It is clear that only at arrival and departure epochs something interesting happens; any time moment in between arrivals and departures can be neglected. Hence, in our simulator we prefer not to keep track of the entire time line, but just of the relevant epochs. If we can do this, we can jump from event to event.

Clearly, we want to follow the sequence of events in the correct order of time. For this we use an *event stack*. To see how this works, it is best to consider a simple example first, and then extend to more difficult situations.

Suppose we have 4 students, and we like to sort them in increasing order of age. One way to do this is to insert them into a list, but the insertion should respect the ordering right away. For this very general problem a number of efficient data structures have been developed, and one of these is the *heap queue*.

Exercise 4.1. Search the web on `python` and `heap queue`. Study the examples and write some code to sort the students Jan(21), Pete(20), Clair(18), Cynthia(25).

Exercise 4.2. Extend the code of the previous exercise such that we can include more information with the ages than just the name, for instance, also the brand of their mobile phone.

Exercise 4.3. It may seem that we have just solved a simple toy problem, but this is not true. In fact, we have established something of real importance: heap queues form the core functionality of (nearly) all discrete event simulators used around the world. To help you understand this fact, sketch how to use heap queues to simulate a queueing process.

4.2 G/G/1 queue

In this section we will set up a simulator for the *G/G*/1 queue with an event stack. Let us work in steps toward the final simulator.

We start with some basic imports, initialize the stack, and define two numbers `ARRIVAL` and `DEPARTURE` that will be used to tag the type of event that will be popped from the event stack. So, make a new file, and type the following at the top of it.

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

np.random.seed(3)

ARRIVAL = 0
DEPARTURE = 1

stack = []  # this is the event stack
```

Note again that we fix the seed so that we get the same random numbers while testing our code.

First of all we need jobs with arrival times and service times. The easiest way to handle jobs in a simulator is by means of a class, as in the code below.

```python
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
```

```python
    def waiting_time(self):
        return self.sojourn_time() - self.service_time

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

A class has a number of *attributes*, such as `self.arrival_time`, to capture its state and a number of functions, such as `def waiting time(self)`, to compute specific information that relates to the class[9]. We initialize the arrival time and service time to zero. Then we store the departure time and queue length for a statistical analysis at the end of the simulations. Finally, we have functions to compute the waiting time and the sojourn time; the `repr` function is used to print the job. Note that our naming of functions also acts as documentation of what the functions do. Note also that member variables and functions in python start with the word `self`; this is to distinguish the (value of the) member variables from variables with the same name but lying outside the scope of the class.

Classes are extremely useful programming concepts, as it enables you to organize state (that is, attributes) and behavior (that is, functions that apply to the attributes) into logical components. Moreover, classes can offer functionality to a programmer without the programmer needing to understand how this functionality is built. Classes offer many more advantages such as inheritance, but we will not discuss that here.

Once we have a class, making an object is simple with this code.[10]

```python
job = Job()
job.arrival_time = 3
job.service_time = 2
```

Exercise 4.4. Make 10 jobs with exponentially distributed inter-arrival times, with $\lambda = 2$, and exponentially distributed service times with $\mu = 3$. Put these jobs on an event stack, and print them in order of arrival time. Tag the events with the job, and event type (which is an arrival).

Now that we know how to make jobs and put them on an event stack, we can make a plan to simulate the $G/G/1$ queue.

---

[9]In python, functions belonging to a class are called 'methods' or 'member functions'.

[10]The wording is important here. Objects are instances of classes. As an example: Albert Einstein was a human being. Here, 'Einstein' is the object, and 'human being' is a class.

Replace the while loop of the previous exercise by the code below. Observe that we split events into two types[11]: arrivals and departures.

```
while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
```

Exercise 4.5. With the above idea to make a while loop, make a list of things that have to take place at an arrival event (in particular, what should happen when the server is busy at an arrival epoch, what should happen if the server is idle?) and at a departure event (in particular, what if the queue is empty, or not empty)? Turn your ideas into code and see whether you can get your simulator working. (It's not a problem if you spend some time on this; you will learn a lot in the process.)

4.3 Testing

As a general observation, testing code is very important. For this reason, before we can use our simulator, we should apply it to some cases that we can analyze with theory.

Exercise 4.6. Run a simulation for the $M/M/1$ queue with $\lambda = 2$ and $\mu = 3$ with 100 jobs and compute the average queue length. Then extend to 1000, and then to 10^5. Compare it to the theoretical expected queue length of the $M/M/1$ queue at arrival moments.

Exercise 4.7. Thus, let us get further confidence in our $G/G/1$ simulator by specializing it to the $D/D/1$ queue. Check the documentation of the `uniform` distribution in `scipy.stats` to see what the following code does

```
from scipy.stats import uniform

F = uniform(3, 0.00001)
G = uniform(2, 0.00001)
```

Then use this in our simulation. What happens? What happens if you reverse the 2 and the 3 in F and G?

---

[11]The word `type` is a reserved word in python, hence I use `typ` instead.

4.4 The check-in process at an airport

We now have a simulator, and we tested it (although not sufficiently well to use it for real applications). We can now start using it. A simple application is to analyze the check-in process at an airport. For ease we constrain the case here to a single server desk. In the next tutorials we will extend this to more realistic scenarios—but that will require extra work. We assume that demand arrives as a Poisson process with rate $\lambda = 1/3$ per minute, and that the service distribution is $U[1,3]$ minute.

Exercise 4.8. Realize that this case can be modeled as an $M/G/1$ queue. Implement the Pollaczek-Khinchine equation in python and use this to compute the expected queue length.

Exercise 4.9. Now estimate the average queue length with the $G/G/1$ simulator. Compare it to the theoretical result of the previous exercise. (Observe that this is yet another test for our $G/G/1$ implementation.)

Exercise 4.10. For the computation of the mean queue length we actually do not need a simulator; the PK-formula suffices. If, however, we are interested in the distribution of the queue length, then we really need the simulator. There are no simple closed-form expressions for the queue length distribution o the $M/G/1$ queue. With our simulator, this is simple.

Use the following code to obtain a quick and dirty overview of the queue length distribution.

```python
from collections import Counter

c = Counter([j.queue_length_at_arrival for j in served_jobs])
print("Queue length distributon, sloppy output")
print(c)
```

What is your opinion? What would you change to the system?

So, once again, we can use simulation to analyze a practical case and come up with concrete recommendations on how to improve the system. It would be interesting to change the demand process such that the arrival rate becomes time-dependent. That would make the case more realistic, and would make the simulator more useful. In fact, the theoretical models nearly always assume that the arrival and service time distribution are constant. When these become dynamic, simulation is the way to go. However, let's stop here.

4.5 Extensions

Here are some final, more general, questions to deepen your understanding of queues systems and simulation.

Exercise 4.11. Up to now we studied the $G/G/1$ FIFO (First In First Out) queue. How to change the code to simulate a LIFO (Last In First Out) queue?

Exercise 4.12. In a priority queue jobs belong to a priority class. Jobs with higher priority are served before jobs with lower priority, and within one priority class jobs are served in FIFO sequence. A concrete example of a priority queue is the check-in process of business and economy class customers at airports. How would you build a $G/G/1$ priority queue?

Exercise 4.13. We implemented the queue as a python list. Why is a deque[12] a more efficient data structure to simulate a queue? Why is it important to know the efficiency of the data structures and algorithms you use?

Exercise 4.14. Recall that in Exercise 4.4 we built all jobs before the queueing simulation starts. Why is this not a good decision? Note that in the simulation of $G/G/c$ queue below we will generate jobs only when needed.

4.6 Summary

Exercise 4.15. What have you learned in this tutorial?

---

[12]Search the web to understand what this is.

5 Simulation of the $G/G/c$ queue in continuous time

The goal of this tutorial is to generalize the simulator for the $G/G/1$ to a $G/G/c$ queue. You will see that this is rather easy, once you have the right idea. We will organize the entire simulator into a single class so that our code is organized and we will develop a method to set up structured tests[13] Once we have a working simulator we apply it to the check-in process of the airport.

5.1 The simulator

Exercise 5.1. In the $G/G/1$ we have just one server which is busy or not. Generalize the `handle_arrival` function of the $G/G/1$ queue such that it can cope with multiple servers.

Exercise 5.2. Likewise, generalize the `handle_departure` function of the $G/G/1$ queue such that it can cope with multiple servers.

Exercise 5.3. Can you think of some sort of property of the $G/G/c$ queue that must be true at all times? It is important to use such properties as tests while developing.

Exercise 5.4. The design of the simulation of $G/G/1$ queue is not entirely satisfactory. For instance, if we want to run different experiments, we have to clean up old experiments before we can start new ones. The current design is not very practical for this. It is better to encapsulate the state and its behavior (i.e., functions) of the simulator into one class. Build a class that simulates the $G/G/c$ queue; use the ideas of the $G/G/1$ simulator.

It is not problem when you spend considerable time on this exercise. Most of the following exercises are simple numerical experiments, which should take little time. So, just try and see how far you get. All code is in the solution.

Exercise 5.5. How can we instantiate the `GGc` simulator, i.e., make an object of a class? How can we run a simulation with exponential inter-arrival times with $\lambda = 2$, exponential service times with $\mu = 1$, $c = 3$, and 10 jobs?

---

[13]Structured testing is enormously important when you use code for real applications. (I suppose you prefer all code used in self-driving cars to be tested.) If you are interested, check the web on `python unittest` and 'test-driven coding'. With test-driven coding you first write down a hierarchy of many simple tests that your program is supposed to pass. Then you build your program until the first test pass. Then you implement the next step until the second test pass too, and so on.

5.2 Testing

Exercise 5.6. Test the simulator with deterministic inter-arrival times, for instance a job that arrives every 10 minutes, and each job takes 1 hour of service. Check the departure process for $c = 1$, $c = 2$, $c = 5$, $c = 6$, and $c = 7$.

Exercise 5.7. For testing purposes, implement Sakasegawa's formula. BTW, why should you use Sakasegawa's formula?

Exercise 5.8. Test the code for the $M/M/1$ queue with $\lambda = 0.8$ and $\mu = 1$. Compute the mean waiting time in queue and compare this to the theoretical value. Test it first for run length $N = 100$, then for $N = 10000$. Think about how to organize your tests.

Exercise 5.9. Now test it for the $M/D/1$ queue with $\lambda = 0.9$ and $S \equiv 1$.

Exercise 5.10. With our simulation we can also check the quality of Sakasegawa's approximation. Try this for the $M/D/2$ queue $\lambda = 1.8$ and $S \equiv 1$.

5.3 The check-in process at an airport

We again analyze the check-in process at an airport, but we make it more realistic than the analysis of Section 4.4 in that we now allow for multiple check-in desks.

Assume that a flight departing at 11 am consists of $N = 300$ people. For ease, assume that all N customers arrive between 9 and 10. Suppose that the check-in of a job is $U[1,3]$ distributed.

Exercise 5.11. Model the arrival process.

The queueing problem is evidently to determine the required number of servers c such that the performance is acceptable.

Exercise 5.12. What performance measures are relevant here?

Exercise 5.13. Define the offered load as $\lambda \mathsf{E}[S]$ and the load as $\rho = \lambda \mathsf{E}[S]/c$. Is it important for the check-in desk that $\rho < 1$?

Exercise 5.14. Why do we need simulation to analyze this case?

Exercise 5.15. Search for a good c.

Exercise 5.16. What do you think of the results? If you are not satisfied, propose and analyze an improvement.

Exercise 5.17. Relate the results of these simulations to you traveling experiences.

5.4 Summary

Exercise 5.18. What have you learned in this tutorial? Can you extend this work to more general cases?

6 Airport Check-in process

As the simulation of this case is somewhat more difficult than the *G/G/c* queue, testing is also more important. For this reason we followed a *test-driven* procedure. First we design a number of test cases, from very simple to a bit less simple. Then we started building and testing until the first test case passed. Then we tested with the second case, the third, and so on, until all the tests passed. And, indeed, along the way we caught numerous mistakes, from missing `if` statements (a very big bug) to simple typos.

Exercise 6.1. Make a list of extremely simple cases, so simple in fact that you can check the outcome with no effort at all, to a bit harder, but such that you have theoretical tools to validate the outcome.

This time we follow

We developed quite a large number of simple cases to test our implementation.

When we use the *M/M/*1 simulation as a test case, we need to have an idea of what variance we can expect between, for instance, the theoretical expected waiting time and the mean waiting time obtained by simulation.

1. Service distribution of the business and economy jobs are different.

7 Hints

h.2.16 Note that the shop sees the combined arrival process, so first find a way to merge the arrival times of the customers into one arrival process as observed by the shop. Then convert this into inter-rival times at the shop.

h.3.4 For the empirical distribution function you can use the code of Exercise 2.7. Before looking it up, try to recall how the cdf is computed.

h.3.10 As a hint, you need a state variable to track whether the extra servers are present or not. The solution shows the code.) Analyze the effect of the threshold at 20; what happens if you set it to 18, say, or 30? What is the effect of the number of extra servers; what if you would add 3 instead of 2?

h.4.6 You might want to check Exercise 4.4 to see how to generate the jobs.

h.4.8 Build a function with arguments λ and G. From from `scipy.stats` import `uniform`. (Read the docs to see how to build the uniform distribution in `scipy.stats`.) Then use `G.var()` and `G.mean()` to compute c_e^2. Then use Little's law to convert the expected waiting time to the expected queue length.

h.4.9 Reset to all data and clear all lists.

h.4.12 We have used a good data structure already. How should this be applied?

h.4.13 Search the web on `python` and `deque`.

h.5.1 Introduce a `num_busy` variable that keeps track of the number of busy servers. What happens if a job arrives and this number is less than c; what if this number is equal to c?

h.5.2 Again, use the `num_busy` variable.

h.5.3 There must be a relation between the queue length and the number of busy servers.

h.5.7 Follow the implementation of Exercise 4.8.

h.5.8 Implement the test in a function so that you can organize all your setting in one clean environment. We can use this below for more general cases.

h.5.9 Implement the deterministic distribution as `uniform(1, 0.00001)`.

h.5.11 Most people arrive independently. What is the arrival rate?

h.5.13 Think hard about the context. Is this queueing process stationary? Is the arrival process stationary?

h.5.15 For this it is best to write two functions. The first function simulates a scenario for a fixed set of parameters and computes the performance measures. The second function calls the simulator over a suitable range for c and prints the KPIs. Like this, we use the second function to fix all parameters so that we always can retrieve the exact scenarios we analyzed.

h.6.1 Suppose there is only a server for business customers and also only business customers? What should happen? Suppose further that the inter-arrival times are constant and 1 minutes and service times constant and equal to 0.5 minute? What if the service times are 2 minutes and 10 business customers arrive? What if all 10 customers arrive at time 0; when should the last leave? What if we only have economy customers but only the business server? And so on. Ensure you design cases you can check by hand. These simple tests will catch many, many errors. As a general rule, invest in such tests.

8 Solutions

s.2.1   1. Generate realizations of a uniformly distributed random variable representing the inter-arrival times of one customer.

2. Plot the inter-arrival times.

3. Compute the (empirical) distribution function of the simulated inter-arrival times.

4. Plot the (empirical) distribution function.

5. Generate realizations of uniformly distributed random variables representing the inter-arrival times of multiple customers, e.g., 3.

6. Compute the arrival times for each customer.

7. Merge the arrival times for all customers. This is the arrival process as seen by the shop.

8. Compute the inter-arrival times as seen by the shop.

9. Plot these inter-arrival times.

10. Compare to the exponential distribution function with a suitable arrival rate λ.

s.2.2 We put the algorithm in a function so that we can use it later. The algorithm is useful to study, but it has some weak points. In the exercises below we will repair the problems.

```python
def cdf(a):
    a = sorted(a)
    m, M = int(min(a)), int(max(a))+1
    # Since we know that a is sorted, this next line
    # would be better, but less clear perhaps:
    # m, M = int(a[0]), int(a[-1])+1

    F = dict() # store the function i \to F[i]
    F[m-1]=0   # since F[x] = 0 for all x < m
    i = 0
    for x in range(m, M):
        F[x] = F[x-1]
```

```
        while i< len(a) and a[i] <= x:
            F[x] += 1
            i += 1

    # normalize
    for x in F.keys():
        F[x] /= len(a)

    return F
```

Now run this to see the result.

```
F = cdf(a)
print(F)
```

s.2.3 We have to guess the support of F (the set of points where F makes the jumps) upfront, and we concentrated the support on the integers. However F makes jumps at floats, for instance at 3.2.

s.2.4 In the answer we let the computer do all the work.

```
I = range(0, len(a))
s = sorted(a)
plt.plot(I, s)
plt.show()
```

s.2.5 Here is one way. Note that we already imported `matplotlib`, so we don't have to that again.

```
def cdf(a):
    y = range(1, len(a)+1)
    y = [yy/len(a) for yy in y] # normalize
    x = sorted(a)
    return x, y

x, y = cdf(a)

plt.plot(x, y)
plt.show()
```

s.2.6 The reason is that at s_1 the first observation occurs. Hence, F should make a jump of at least one at s_1. Next, the `range` function works up to, but not including, its second argument. Hence (in code), `range(10)[-1]/10 = 0.9`, that is, the last element `range(10)[-]` of the set of numbers $0, 1, \ldots, 9$ is not 10. Hence, when we extend the range to `len(a)+1` we have a range up to and including the element we want to include.

s.2.7 This code is much, much faster, and also very clean. Note that we normalize y right away.

```python
def cdf(a):
    y = np.arange(1, len(a)+1)/len(a)
    x = np.sort(a)
    return x, y
```

s.2.8 With the `drawstyle` option:

```python
plt.plot(x, y,  drawstyle = 'steps-post')
plt.show()
```

But now we still have vertical lines. To remove those, we can use `hlines`.

```python
y = range(0, len(a)+1)
y = [yy/len(a) for yy in y] # normalize
s = sorted(a)
left = [min(a)-1] + a
right = a + [max(a)+1]

plt.hlines(y, left, right)
plt.show()
```

There we are!.

s.2.9 Here it is.

```python
def cdf(X):
    # remove multiple occurrences of the same value
    unique, count = np.unique(np.sort(X), return_counts=True)
    x = unique
    y = count.cumsum()/count.sum()
    return x, y

x, y = cdf(a)
```

s.2.10 Copy this code and run it.

```python
from scipy.stats import uniform

# fix the seed
scipy.random.seed(3)

# parameters
L = 3    # number of inter-arrival times

G = uniform(loc=4, scale=2) # G is called a frozen distribution.
a = G.rvs(L)
print(a)
```

s.2.11 Add this code to the other code and run it.

```python
N = 1 # number of customers
L = 300
a = G.rvs(L)

plt.hist(a, bins=L/20, label="a")
plt.title("N = {}, L = {}".format(N, L))
plt.legend()
plt.show()
```

s.2.12 Add this to the other code and run it.

```python
x, y = cdf(a)
plt.plot(x,y,  label="d")
plt.legend()
plt.show()
```

s.2.13 Add this to the other code and run it.

```python
def KS(X, F):
    # Compute the Kolmogorov-Smirnov statistic where
    # X are the data points
    # F is the theoretical distribution
    support, y = cdf(X)
    y_theo = np.array([F.cdf(x) for x in support])
    return np.max(np.abs(y-y_theo))
```

```
print(KS(a, G))
```

s.2.14 Add this to the other code and run it. Since the mean inter-arrival time is 5, take $\lambda = 1/5$.

```python
from scipy.stats import expon

labda = 1./5 # lambda is a function in python
E = expon(scale=1./labda)
print(E.mean()) # to check that we chose the right scale
print(KS(a, E))
```

s.2.15 Add this to the other code and run it.

```python
x, y = cdf(a)
dist_name = "U[4,6]"
def plot_distributions(x, y, N, L, dist_name):
    # plot the empirical cdf and the theoretical cdf in one figure
    plt.title("X ~ {} with N = {}, L = {}".format(dist_name,N, L))
    plt.plot(x, y, label="empirical")
    plt.plot(x, E.cdf(x), label="exponential")
    plt.legend()
    plt.show()

plot_distributions(x, y, N, L, dist_name)
```

It is pretty obvious why these graphs must be different: we compare a uniform and an exponential distribution.

s.2.16 The following steps in code explain the logic.

```python
a=[4, 3, 1.2, 5]
b=[2, 0.5, 9]

def compute_arrivaltimes(a):
    A=[0]
    i = 1
    for x in a:
        A.append(A[i-1] + x)
```

```
        i += 1

    return A

A = compute_arrivaltimes(a)
B = compute_arrivaltimes(b)


times = [0] + sorted(A[1:] + B[1:])
print(times)

shop = []
for s, t in zip(times[:-1], times[1:]):
    shop.append(t - s)

print(shop)
```

s.2.18 Add this to the other code and run it.

```
N, L = 3, 100
a = superposition(G.rvs((N, L)))

E = expon(scale=1./(N*labda))
print(E.mean())

x, y = cdf(a)
dist_name ="U[4,6]"
plot_distributions(x, y, N, L, dist_name)

print(KS(a, E)) # Compute KS statistic using the function defined earlie
```

s.2.19 Add this to the other code and run it.

```
N, L = 10, 100
a = superposition(G.rvs((N, L)))

E = expon(scale=1./(N*labda))

x, y = cdf(a)
dist_name ="U[4,6]"
```

```
plot_distributions(x, y, N, L, dist_name)

print(KS(a, E))
```

This is great. For just $N = 10$ we see that the exponential distribution is a real good fit.

s.2.20 Add this to the other code and run it.

```
from scipy.stats import norm

N, L = 10, 100

N_dist = norm(loc=5, scale=1)
a = superposition(N_dist.rvs((N, L)))
x, y = cdf(a)
dist_name = "N(5,1)"
plot_distributions(x, y, N, L, dist_name)

print(KS(a, E))
```

Clearly, whether the distribution of inter-arrival times of an individual customer are uniform, or normal, doesn't really matter. In both cases the exponential distribution is a good model for what the shop sees. We did not analyze what happens if we would merge customers with normal distribution and uniform distribution, but we can suspect that in all these cases the merged process converges to a set of i.i.d. exponentially distributed random variables.

s.2.21 Since $\sigma = \mu = 5$, about 15% of the 'inter-arrival' times should be negative. This is clearly impossible.

s.2.22 Here are some ideas you should have learned.

1. Algorithmic thinking, i.e., how to chop up a computational challenge into small steps.

2. How to efficiently compute the empirical distribution function

3. The Kolmogorov-Smirnov statistic

4. The empirical distribution of a merged inter-arrival arrival process converges, typically, super rapidly to an exponential distribution. Thus, inter-arrival times at shops, hospitals and so on, are often very well described by an exponential distribution with suitable mean.

5. This convergence is not so sensitive to the distribution of the inter-arrival times of a single customers. For the shop, only the population matters.

6. Using functions (e.g., `def compute(a)`) to document code (by the function name), hide complexity, and reuse code so that it can be applied multiple times. Moreover, defining functions is in line with the extremely important Don't-Repeat-Yourself (DRY) principle.

7. Coding skills: python, `numpy` and `scipy`.

s.3.1 We need a few imports.

```python
import numpy as np
import scipy
from scipy.stats import poisson
import matplotlib.pyplot as plt

scipy.random.seed(3)


def compute_Q_d(a, s, q0=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0 # starting level of the queue
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = Q[i-1] + a[i] - d[i]

    return Q, d
```

One of the nicest things of python is that the real code and pseudo code resemble each other so much.

s.3.2 Here is the complete code in case you have messed things up.

```python
import numpy as np
import scipy
from scipy.stats import poisson
import matplotlib.pyplot as plt

scipy.random.seed(3)
```

```
def compute_Q_d(a, s, q0=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0 # starting level of the queue
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = Q[i-1] + a[i] - d[i]

    return Q, d


labda, mu, q0, N = 5, 6, 0, 100
a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)
print(a.mean(), a.std())
```

s.3.4 This it the code.

```
print(d.mean())

x, F = cdf(Q)
plt.plot(x, F)
plt.show()
```

s.3.5 The mean number of departures must be (about) equal to the mean number arrivals per period. Jobs cannot enter from 'nowhere'.

s.3.6 The queue length drains at rate $\mu - \lambda$ when q_0 is really large. Thus, for such settings, you might just as well approximate the queue length behavior as $q(t) = q_0 - (\mu - \lambda)t$, i.e., as a deterministic system.

s.3.8 You should observe that the variability of the queue length process decreases..

s.3.9 For our experiments it is about 20% extra.

s.3.10 Here is one way.

```python
def compute_Q_d(a, q0=0, mu=6, threshold=20, extra=2):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False # extra employees are not in
    for i in range(1, len(a)):
        rate = mu + extra if present else mu # service rate
        s = poisson(rate).rvs()
        d[i] = min(Q[i-1],s)
        Q[i] = Q[i-1] + a[i] - d[i]
        if Q[i] == 0:
            present = False # send employee home
        elif Q[i] >= threshold:
            present = True # hire employee for next period

    return Q, d
```

Note that this code runs significantly slower than the other code. This is because we now have to call `poisson(mu).rvs()` in every step of the for-loop. We could make the program run faster by using the `scipy` library to generate N random numbers in one step outside of the for loop and then extract numbers from there when needed. However, for clarity we chose not to do so.

s.3.11 Here is an example. What can be the meaning of `np.inf`?

```python
def compute_Q_d(a, s, q0=0, b=np.inf):
    # b is the blocking level.
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    for i in range(1, len(a)):
        d[i] = min(Q[i-1], s[i])
        Q[i] = min(b, Q[i-1] + a[i] - d[i])

    return Q, d
```

```python
N = 10000
labda = 5
mu = 6
```

```
q0 = 0

a = poisson(labda).rvs(N)
s = poisson(mu).rvs(N)

Q, d = compute_Q_d(a, s, q0, b=15)
print(Q.mean(), Q.std())

x, F = cdf(Q)
plt.plot(x, F)
plt.show()
```

s.3.12 Here is all the code.

```
num_jobs = 10000
labda = 0.3
mu = 1
q0 = 0
N = 100 # threshold

h = 1
p = 5
S = 500


def compute_cost(a, q0=0):
    d = np.zeros_like(a)
    Q = np.zeros_like(a)
    Q[0] = q0
    present = False  # extra employee is not in.
    queueing_cost = 0
    server_cost = 0
    setup_cost = 0
    for i in range(1, len(a)):
        if present:
            server_cost += p
            s = poisson(mu).rvs()
        else:
            s = 0
```

```
        d[i] = min(Q[i-1], s)
        Q[i] = Q[i-1] + a[i] - d[i]
        if Q[i] == 0:
            present = False  # send employee home
        elif Q[i] >= N:
            if present == False:
                present = True # server is switched on
                setup_cost += S
        queueing_cost += h * Q[i]

    print(queueing_cost, setup_cost, server_cost)

    total_cost = queueing_cost + server_cost + setup_cost
    num_periods = len(a) - 1
    average_cost = total_cost / num_periods
    return average_cost

a = poisson(labda).rvs(num_jobs)
av = compute_cost(a, q0)
print(av)
```

After a bit of experimentation, we see that $N = 15$ is quite a bit better than $N = 100$.

s.3.13 Some important points are the following.

1. Making a simulation requires some ingenuity, but is often not difficult

2. With simulation it becomes possible to analyze many difficult queueing situations. The mathematical analysis of often much harder, if possible at all.

3. We studied the behavior of queues under certain control policies, typically policies that change the service rate as a function of the queue length.

   An interesting extension is to incorporate time-varying demand. In many systems, such as supermarkets, the demand is not constant over the day. In such cases the planning of servers should take this into account.

s.4.1 If you found, and read, the appropriate web page, you must have ended up with this code.

```
from heapq import heappop, heappush
```

```
stack = []

heappush(stack, (21, "Jan"))
heappush(stack, (20, "Piet"))
heappush(stack, (18, "Klara"))
heappush(stack, (25, "Cynthia"))

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

Pushing puts things on the stack in a sorted fashion, popping takes things from the stack. First we put the students on the stack. To print, we remove items from the stack until it is empty.

s.4.2 We can make the tuples, i.e., the data between the brackets, just longer.

```
from heapq import heappop, heappush

stack = []

heappush(stack, (21, "Jan", "Huawei"))
heappush(stack, (20, "Piet", "Apple"))
heappush(stack, (18, "Klara", "Motorola"))
heappush(stack, (25, "Cynthia", "Nexus"))

while stack:
    age, name, phone = heappop(stack)
    print(age, name, phone)
```

s.4.3 For a queueing process we can start with putting a number of job arrivals on the stack and label these events as 'arrivals'. Whenever a service starts, compute the departure time of a job, and put this departure moment on the stack. Label this event as a 'departure'. Then move to the next event. Since the event stack is sorted in time, the event at the head of the stack is the first moment in time something useful happens.

More generally, a discrete-time stochastic process moves from event to event. At an event certain actions have to be taken, and these actions may involve the generation of new events or the removal of old events. The new events are put on the

stack, the obsolete ones removed, and then the simulator moves to the first event on the stack.

s.4.4 One way is like this.

```python
labda = 2.
mu = 3.
rho = labda/mu
F = expon(scale=1./labda)  # interarrival time distributon
G = expon(scale=1./mu)   # service time distributon

num_jobs = 10

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


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

s.4.5 We need a server object to keep track of the state of the server, and we also need a list to queue the jobs.

```python
class Server:
    def __init__(self):
        self.busy = False

server = Server()
queue = []
served_jobs = []  # used for statistics

def start_service(time, job):
    server.busy = True
    job.departure_time = time + job.service_time
```

```
        heappush(stack, (job.departure_time, job, DEPARTURE))

def handle_arrival(time, job):
    job.queue_length_at_arrival = len(queue)
    if server.busy:
        queue.append(job)
    else:
        start_service(time, job)

def handle_departure(time, job):
    server.busy = False
    if queue: # queue is not empty
        next_job = queue.pop(0) # pop oldest job in queue
        start_service(time, next_job)

while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
        served_jobs.append(job)
```

s.4.6 Now we see that we needed to store the jobs in `served_jobs`. Set `num_jobs=100` in the code of Exercise 4.4. Put the following code at the end of the simulator to compute the statistics.

```
tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
av_queue_length = tot_queue/len(served_jobs)
print("Theoretical avg. queue length: ", rho*rho/(1-rho))
print("Simulated avg. queue length:", av_queue_length)

tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
av_sojourn_time = tot_sojourn/len(served_jobs)
print("Avg. sojourn time:", av_sojourn_time)
```

Now run the simulator.

You should see that you need a real large amount of jobs to obtain a good estimate for the (theoretical) expected queue length[14].

---

[14]I have to admit that I don't understand why (at least in my simulation) I need about $1e5$ jobs to get

s.4.7 With F=uniform(3, 0.00001) job inter-arrival times are nearly 3. Likewise, the service times are nearly 2. Hence, arriving customers will always see an empty queue, implying that the average queue length upon arrival equals zero. When you reverse the 2 and 3 a queue should build up.

s.4.8 First we compute the average waiting time. Then we use Little's law to convert the waiting time to the expected queue length.

```python
def pollaczek_khinchine(labda, G):
    ES = G.mean()
    rho = labda*ES
    ce2 = G.var()/ES/ES
    EW = (1.+ce2)/2 * rho/(1-rho)*ES
    return EW


labda = 1./3
F = expon(scale=1./labda)  # interarrival time distributon
G = uniform(1, 2)

print("PK: ", labda*pollaczek_khinchine(labda,G))
```

s.4.9 To ensure that we do not keep old data, we have to reset all lists.

```python
print("Simulation of check-in process, M/G/1 with G=U[2,5]")


stack = []  # this is the event stack
queue = []
served_jobs = []  # used for statistics

job = Job()

num_jobs = 100000

time = 0
for i in range(num_jobs):
    time += F.rvs()
```
_____

a good estimate.

```
    job = Job()
    job.arrival_time = time
    job.service_time = G.rvs()
    heappush(stack, (job.arrival_time, job, ARRIVAL))

while stack:
    time, job, typ = heappop(stack)
    if typ == ARRIVAL:
        handle_arrival(time, job)
    else:
        handle_departure(time, job)
        served_jobs.append(job)

tot_queue = sum(j.queue_length_at_arrival for j in served_jobs)
av_queue_length = tot_queue/len(served_jobs)
print("Theo avg. queue length: ", labda*pollaczek_khinchine(labda,G))
print("Simu avg. queue length:", av_queue_length)

tot_sojourn = sum(j.sojourn_time() for j in served_jobs)
av_sojourn_time = tot_sojourn/len(served_jobs)
print("Theo avg. sojourn time: ", pollaczek_khinchine(labda,G) + G.mean(
print("Simu avg. sojourn time:", av_sojourn_time)
```

s.4.10 I ran the code for $n = 100000$ and got this

```
 Counter({0: 37134, 1: 16558, 2: 12517, 3: 9070, 4: 6725, 5: 4785, 6: 35

   8: 2047, 9: 1433, 10: 1095,  11: 818, 12: 545, 13: 401, 14: 260,

   15: 178, 16: 116, 17: 72, 18: 49, 19: 27, 20: 13, 21: 6, 22: 5})
```

So, about 2% see a queue that is longer than 10, and about 10% (which I get by, so-called-eye balling) of the people see a queue that is longer than 5. It seems that the service capacity is too small, assuming that the customers have a flight to catch.

s.4.11 This is really easy: change the line `queue.pop(0)` by `queue.pop()`.

s.4.12 As in the LIFO example, we only have to change the data structure. First, we give each job an extra attribute corresponding to its priority. Then we use a heap to store the jobs in queue. Specifically, change the line with `queue.append(job)` by

`heappush(queue, (job.priority, job))`

and `queue.pop(0)` by `heappop(queue)`.

s.4.13 In a python list the `pop(0)` function is an $O(n)$ operation, where n is the number of elements in the list. In a deque appending and removing items to either end of the deque is an $O(1)$ operation.

  When you do large scale simulations, involving many hours of simulation time, all inefficiencies build up and can make the running time orders of magnitude longer. A famous example is the sorting of numbers. A simple, but stupid, sorting algorithm is $O(n^2)$ while a good algorithm is $O(n \log n)$. The sorting time of 10^6 numbers is dramatically different.

  For our code, add the line `from collections import` deque and replace `queue = []` by `queue = deque()`, and `queue.pop(0)` by `queue.popleft()`.

s.4.14 Typically, generating all jobs at the start is not a real good idea. When running large simulations the amount of computer memory required to store all this data grows out of hand. Also our `served_jobs` list is not memory efficient.

s.4.15 Topics learned.

1. Efficient data structures such as heap queues. In general, it is important to have some basic knowledge of algorithmic complexity.

2. Event stacks to organize the tracking of events in time. With the concept of event stack, you now know how discrete event simulators work.

3. The simulation of the $G/G/1$ queue

s.5.1 If the number of busy servers is less than c, a service can start. Otherwise a job has to queue. If a service starts, increase `num_busy` by one.

s.5.2 At a departure, a server becomes free, hence decrease `num_busy` by one. If there are jobs in queue, start a new service.

s.5.3 Of course, the number of busy servers cannot be negative or larger than c. The queue length cannot be negative. Finally, it cannot be that the queue length is positive and the number of busy servers is less than c.

s.5.4 Here is the entire class. Read it carefully.

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

np.random.seed(3)

ARRIVAL = 0
DEPARTURE = 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):
        return self.sojourn_time() - self.service_time

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


class GGc:
    def __init__(self, F, G, c, run_length):
        self.F = F
        self.G = G
        self.c = c
        self.num_busy = 0    # number of busy servers
```

```python
        self.run_length = run_length
        self.stack = []  # event stack
        self.queue = deque()
        self.served_jobs = set()

    def handle_arrival(self, time, job):
        self.generate_new_arrival(time)
        job.queue_length_at_arrival = len(self.queue)
        if self.num_busy < self.c:
            self.start_service(time, job)
        else:
            self.queue.append(job)

    def generate_new_arrival(self, time):
        if self.run_length > 0:
            self.put_new_arrival_on_stack(time)
            self.run_length -= 1

    def put_new_arrival_on_stack(self, time):
        job = Job()
        job.arrival_time = time + self.F.rvs()  # new arrival time
        job.service_time = self.G.rvs()   # new service time
        heappush(self.stack, (job.arrival_time, job, ARRIVAL))

    def start_service(self, time, job):
        self.num_busy += 1  # server becomes busy.
        job.departure_time = time + job.service_time
        heappush(self.stack, (job.departure_time, job, DEPARTURE))

    def handle_departure(self, time, job):
        self.num_busy -= 1
        self.served_jobs.add(job)
        if self.queue:  # not empty
            next_job = self.queue.popleft()
            self.start_service(time, next_job)

    def consistency_check(self):
        if ( self.num_busy < 0 or self.num_busy > self.c \
```

```
                or len(self.queue) < 0 or \
                (len(self.queue) > 0 and self.num_busy < self.c) ):
                print("there is something wrong")
                quit()

    def run(self):
        time = 0
        self.put_new_arrival_on_stack(time)

        while self.stack:   # not empty
            time, job, typ = heappop(self.stack)
            # self.consistency_check() # only use when testing.
            if typ == ARRIVAL:
                self.handle_arrival(time, job)
            else:
                self.handle_departure(time, job)
```

s.5.5 First we instantiate, then we run and print.

```
labda = 2
mu = 1
ggc = GGc(expon(scale=1./labda), expon(scale=1./mu), 3, 10)
ggc.run()

print(ggc.served_jobs)
```

s.5.7 Recall that Sakasegawa's formula is exact for the $M/G/1$ queue, hence also for the $M/M/1$ queue.

```
def sakasegawa(F, G, c):
    labda = 1./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
```

s.5.8 In the next function we perform the test. We give this function standard arguments so that we can run the function as a standalone test. As such we want to to contain all parameter values.

```python
def mm1_test(labda=0.8, mu=1, num_jobs=100):
    c = 1
    F = expon(scale=1./labda)
    G = expon(scale=1./mu)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jobs)
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/M/1 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G, c))
    print("Simu avg. waiting time in queue:", avg_wait_in_q)

mm1_test(num_jobs=100)
mm1_test(num_jobs=10000)
```

s.5.9 Here is a function to keep things organized.

```python
def md1_test(labda=0.9, mu=1, num_jobs=100):
    c = 1
    F = expon(scale=1./labda)
    G = uniform(mu, 0.0001)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jobs)
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/D/1 TEST")
    print("Theo avg. waiting time in queue:", sakasegawa(F, G, c))
    print("Simu avg. waiting time in queue:", avg_wait_in_q)

md1_test(num_jobs=100)
md1_test(num_jobs=10000)
```

s.5.10 The code.

```python
def md2_test(labda=1.8, mu=1, num_jobs=100):
    c = 2
```

```python
    F = expon(scale=1./labda)
    G = uniform(mu, 0.0001)

    ggc = GGc(F, G, c, num_jobs)
    ggc.run()
    tot_wait_in_q = sum(j.waiting_time() for j in ggc.served_jobs)
    avg_wait_in_q = tot_wait_in_q/len(ggc.served_jobs)

    print("M/D/2 TEST")
    print("Theoretical avg. waiting time in queue:", sakasegawa(F, G, c)
    print("Simulated avg. waiting time in queue:", avg_wait_in_q)

md2_test(num_jobs=100)
md2_test(num_jobs=10000)
```

s.5.11 Since most people arrive independent of each other, it is reasonable that the arrival process in this period of duration T is Poisson with rate $\lambda = N/T$.

s.5.12 It is reasonable *within the context* that the mean waiting time is less important than the largest waiting time and the fraction of people that wait longer than 10 minutes, say. Note here, *context is crucial* when making models.

s.5.13 There are, by assumption, only arrivals between 9 and 10, hence the arrival process is not stationary, hence the queueing process is not stationary. It is also not a problem if the system is temporarily in overload. In fact, as long as c is such that the performance measures (maximum delay and fraction of people waiting longer than some threshold) are within reasonable bounds, the system works as it should.

s.5.14 We study the dynamics of a queueing process in which the demand grows and than peters out. Such queueing situations are, typically, mathematically intractable. In fact, we finally have come to a point in which we really need simulation.

s.5.15 The first function computes the KPIs, the second runs multiple scenarios.

```python
def intake_process(F, G, c, num):

    ggc = GGc(F, G, c, num)
    ggc.run()

    max_waiting_time = max(j.waiting_time() for j in ggc.served_jobs)
```

```
        longer_ten = sum((j.waiting_time()>=10)for j in ggc.served_jobs)
        return max_waiting_time, longer_ten


def intake_test_1():
    labda = 300/60
    F = expon(scale=1.0 / labda)
    G = uniform(1, 2)
    num = 300

    print("Num servers, max waiting time, num longer than 10")
    for c in range(3, 10):
        max_waiting_time, longer_ten = intake_process(F, G, c, num)
        print(c, max_waiting_time, longer_ten)


intake_test_1()
```

s.5.16 I get these numbers

```
Num servers, max waiting time, num longer than 10
3 138.1697966325396 280
4 82.7792212258584 266
5 54.741743193174834 227
6 39.35432320034931 220
7 22.868514027211248 167
8 15.77953068102207 102
9 12.118348481100451 50
```

Even for 9 servers, 50 people wait longer than 10 minutes. Interestingly, with 9 servers the maximum waiting time is just 12 minutes, so we can at least keep this within bounds.

We could also make a plot of the job waiting time as a function of time, but we skip it here.

Perhaps we should open the check-in desks for a longer time and advise people to come earlier. Let's try 90 minutes instead.

```
def intake_test_2():
    labda = 300/90
```

```python
F = expon(scale=1.0 / labda)
G = uniform(1, 2)
num = 300

print("Num servers, max waiting time, num longer than 10")
for c in range(3, 10):
    max_waiting_time, longer_ten = intake_process(F, G, c, num)
    print(c, max_waiting_time, longer_ten)
```

If you do the experiment, you'll see that this works much better. It is now also very simple to set up a third experiment in which the desks are open during two hours. Then you'll see that 4 servers suffice.

s.5.18 Topics learned.

1. The simulation of the *G/G/c* queue

2. Python classes and, a bit more generally, object oriented programming.

3. Organizing cases and experiments by means of functions. With these functions we can keep a log of what we precisely tested, what parameter values we used and which code we ran. These tests are also useful to show how to actually use/run the code.

Some interesting extensions.

1. Scale up to networks of *G/G/c* queues.

2. In the *G/G/c* queue the assumption is that all servers have the same service rate. In production settings this is typically not the case. There is a queue of jobs, and these jobs are served by machines with different speeds. For instance, old machines may work at a slower rate than new machines.