Sample Path Analysis and Simulation of Stochastic Systems

Nicky D. van Foreest

November 25, 2024

CONTENTS

CONTENTS IN DETAIL

1 INTRODUCTION

1.1 MOTIVATION AND OVERVIEW

Queueing and inventory systems abound, and the analysis, performance evaluation, control and optimization of these stochastic systems are major topics in Operations Research.

For instance, fast food restaurants deal with many difficult queueing and inventory situations. They prepackage hamburgers and put these on shelf so that when a customer arrives, the customer does not have to wait for the product. The life time of such products is quite small, in the order of a few minutes, and when a customer does not show up in time, the product has to be scrapped. The question is here to balance the number of hamburgers to put on stock so that both the probability of scrapping and the probability that a customer has to wait are small. Fast food restaurants keep also an inventory of baked patties, which are not yet turned into hamburgers (with a bun). The patties have a longer life time than prepared hamburgers, but as it requires some work to turn it into a hamburger, it might give a potential waiting time for a customer. Again, the inventory level of these patties need to be controlled. This is still not the end of the matter because the rate of customer demand changes over time, so that the inventory levels of (the different types of) hamburgers and patties need to be time dependent.

This is called make-to-stock production.

Customers that go elsewhere are known as lost sales.

As a second example, service systems, such as hospitals, call centers, courts, have a certain capacity available to serve customers. The performance of such systems is, in part, measured by the total number of jobs processed per year and the fraction of jobs processed within a certain time frame between receiving and closing the job. Here the problem is to organize the capacity such that the sojourn time, i.e., the typical time a job spends in the system, does not exceed some threshold.

Inventory and queueing systems are tightly related by changing perspective. Hamburgers in stock can be interpreted as customers in queue *waiting for service, where the service time is the time between the arrival of two customers that buy hamburgers.*

Clearly, in all these examples, the performance of the stochastic system has to be monitored and controlled. Typically the following performance measures are relevant for queueing systems.

Also known as Key Performance Indicators (KPIs).

1. The fraction of time $p(n)$ that the system contains n customers. In particular, $1 - p(0)$, i.e., the fraction of time the system contains jobs, is important, as this is a measure of the time-average occupancy of the servers, hence related to personnel cost.

2. The fraction of customers $\pi(n)$ that 'see upon arrival' the system with n customers. This measure relates to customer perception and lost sales, i.e., fractions of arriving customers that do not enter the system.

3. The average, variance, and/or distribution of the waiting time.

4. The average, variance, and/or distribution of the number of customers in the system.

Inventory systems have very similar performance measures such that average time items spend on shelf, the fraction of customers served from stock, the fraction of lost sales.

With these examples in mind, let us give an overview of the chapters of this book.

CHAPTER 2 STARTS WITH constructing queueing and inventory systems in discrete time. This serves three goals. First, construction is concrete, so that by specifying the rules to characterize the behavior of the system, you (the reader) develop essential modeling skills. Second, these rules can often be easily implemented in computer code and used to simulate and control actual stochastic systems. Simulation is in general the best way to analyze practical stochastic systems, as realistic systems seldom yield to mathematics. Third, simulation provides us with sample paths of the behavior of the system, but we will also use sample-path arguments to develop the theoretical results of the book.

In Chapter 3 we move on to systems in continuous time and show again how to set up simulation environments to construct sample paths, in particular *discrete event simulation*.

(Nearly?) all professional simulation tools are built on the same principles.

Once it is clear what queueing and inventory theory is about, the stage is set for a more mathematical treatment of such systems. In Chapter 4 we develop some necessary key results. For this, we use sample paths of stochastic processes, and by assuming that such sample paths capture the 'normal' stochastic behavior, we can use the sample paths to estimate the most important performance measures.

In Chapter 5 we use these tools to develop exact models for single-station queueing systems. In our discussions we mostly focus on obtaining an intuitive understanding of the analytical tools.

For most of the proofs and/or more extensive results we refer to the bibliography at the end of the book.

Notwithstanding the power of simulation, it is often hard to obtain structural understanding of the behavior of stochastic systems. Instead, mathematical models, whether exact or approximate, are useful to help reason about and improve such systems. In Chapter 6 we use approximations and general results of probability theory to understand how production and service situations are affected by the system parameters such as service speed, batching rules, and outages.

WHILE THE MAIN text contains many examples and derivations, a considerable number of examples are delegated to exercises. Also, some of these exercises are consistency checks between results derived for different models, thereby providing important relations between various parts of the text. The exercises are not meant to be really easy; they should require (some) work. Hints and solutions to all problems are available at the end of the book.

Note that, while such checks are trivial in principle, the algebra can be quite difficult at times.

IT REMAINS TO discuss the contents of this chapter. Clearly, this Section 1.1 provides some high level motivation why to study stochastic systems, of which queueing and inventory processes are prime examples. Section 1.2 introduces and relates the Exponential distribution and the Poisson distribution, which are, perhaps, the most important distributions to model dynamic stochastic systems. In Section 1.3 we use simulation and some maths to explain why these distribution's are particularly useful to arrival processes of customers or jobs in queueing systems and demands for inventory systems. These amount of computational work is quite involved,

but the computer is our diligent assistant here. In Section 1.4 we discuss the Python code that handles these numerical aspects. Not only will we use this code multiple times in the book, we show how to convert the ideas and notation we are used to think about into an elegant program so that programming becomes an agreeable and intellectually stimulating activity.

TRUE-FALSE QUESTIONS are simple statements that can be true or false; you have to find out the correct alternative.

TF 1.1.1. Consider the next code.

```
a = 8
a += 9
a = 10
a -= 3
```

Claim: `a = 10`

TF 1.1.2. Consider the next code.

```
a = [0, 2, 4, 6]
a[1] += 1
```

Claim: `a = [0, 3, 4, 6]`

TF 1.1.3. Consider the next code.

```
a = {-10: 1, 2: 8}
a[1] += 1
```

Claim: `a = {-10: 1, 1: 1, 2: 8}`.

TF 1.1.4. Consider the next code.

```
from collections import defaultdict

a = defaultdict(int)
a[-8] = 1
a[2] = 8
a[1] += 3
a[2] -= 3
```

Claim: `a = {-8: 1, 1: 3, 2: 5}`.

TF 1.1.5. Consider the next code.

```
from collections import defaultdict

a = defaultdict(int)
a[-8] = 1
a[2] = 8
a[1] += 3
a[2] -= 3
```

Claim: the keys of a are $-8, 1, 2$ and the values $1, 3, 5$.

1.2 EXPONENTIAL AND POISSON DISTRIBUTION

In this section we make a model for the arrival process that is the most useful for queueing and inventory systems. In particular, we show how the exponential and gamma distribution relate to the Poisson distribution, and we use the memoryless property of job inter-arrival times as the motivating point of departure.

Arrivals in queueing systems are often called customers ar jobs, *and* demand *in inventory systems.*

THE TIMESCALES THAT are relevant for our purposes range roughly from minutes to a week. On such timescales it seems reasonable to model inter-arrival times as memoryless. For instance, if X is the inter-arrival time between two patients with a broken arm at the emergency room of a hospital, what can we say about the time the next patient with a broken arm arrives if somebody tells us that an hour earlier a similar patient arrived? Not much, as most of us will agree. Similarly, suppose that the average time between the sales of some shirt in a clothes shop is 5 minutes, and the shop owner informs us that no shirt has been sold the past 7 minutes. Does that knowledge imply that the next shirt must be sold within the next minute? Surely not.

With exceptions, naturally.

In more formal terms we claim that the probability that the next arrival will not occur within t time units from now does not depend on somebody telling us that there wasn't an arrival in the past s time units. In other words, if the person does not tell us anything, we write the probability that no arrival occurs before time t from now as $\mathsf{P}\{X > t\}$. However, if the person tells us at time s there was no arrival during $[0, s]$, we write that the probability that there will not be an arrival for an additional amount time t as $\mathsf{P}\{X > s + t | X > s\}$. If we believe that the information $\{X > s\}$ is immaterial to our probabilistic model, then necessarily $\mathsf{P}\{X > s + t \,|\, X > s\} = \mathsf{P}\{X > t\}$.

Definition 1.2.1. A rv X whose cdf satisfies

$$\mathsf{P}\{X > s + t \,|\, X > s\} = \mathsf{P}\{X > t\}, \quad s, t \geq 0, \tag{1.2.1}$$

cdf := cumulative distribution function

is said to be *memoryless*.

As a counter example, the life span of human beings is not memoryless: take X as the life span of an arbitrary person born in 1820, and $s = 100$ and $t = 99$ years. Then $\mathsf{P}\{X > 99\}$ was small, but not zero, but $\mathsf{P}\{X > 199 \,|\, X > 100\} = 0$.

THE NEXT THEOREM is fundamental.

Theorem 1.2.2. $X \sim \mathrm{Exp}(\lambda)$ iff X is memoryless.

Proof. $\implies$ Observing that $\mathsf{P}\{X \leq t\} = 1 - e^{-\lambda t} \implies \mathsf{P}\{X > t\} = e^{-\lambda t}$,

$$\begin{aligned} \mathsf{P}\{X > s + t \,|\, X > s\} &\overset{1}{=} \frac{\mathsf{P}\{X > s + t\}}{\mathsf{P}\{X > s\}} = \frac{e^{-\lambda(t+s)}}{e^{-\lambda s}} \\ &= e^{-\lambda t} = \mathsf{P}\{X > t\}, \end{aligned}$$

where Step 1 follows from the standard formula for conditional probability and that $\{X > t + s\} \subset \{X > s\}$. Next, $\Longleftarrow$, this is slightly more difficult, so we skip it. □

For a proof, see Yushkevich and Dynkin [1969, Appendix 3] [1.2.6]

Note that for $X \sim \mathrm{Exp}(\lambda)$, the following properties hold:

$$\mathsf{E}[X] = \lambda^{-1}, \qquad \mathsf{V}[X] = \lambda^{-2}.$$

If we have a sequence of inter-arrival times $\{X_i\}_{i=1}^{\infty}$, then we construct arrival times according to the rule

$$A_k = A_{k-1} + X_k, \quad A_0 = 0.$$

When inter-arrival times are iid rvs and distributed as the common rv $X \sim \text{Exp}(\lambda)$, it is clear that the density of the first arrival time is

iid := independent and identically distributed

rv(s) := random variable(s)

$$f_{A_1}(t) = f_X(t) = \lambda e^{-\lambda t} = \lambda \frac{(\lambda t)^0}{0!} e^{-\lambda t},$$

where we add the last term to see how a pattern emerges. Using convolution, for the second arrival time,

$$f_{A_2}(t) = \int_0^t f_{A_1}(s) f_X(t-s)\,\mathrm{d}s = \int_0^t \lambda e^{-\lambda s} \lambda e^{-\lambda(t-s)}\,\mathrm{d}s = \lambda \frac{(\lambda t)^1}{1!} e^{-\lambda t}.$$

With induction it is straightforward to extend the pattern and prove that $A_k \sim \text{Gamma}(\lambda, k)$, i.e.,

$$f_{A_k}(t) = \int_0^t f_{A_{k-1}}(s) f_X(t-s)\,\mathrm{d}s = \lambda \frac{(\lambda t)^{k-1}}{(k-1)!} e^{-\lambda t}. \tag{1.2.2}$$

WHEN THE ARRIVAL times are gamma distributed, the cdf of the number $N(t)$ of jobs arriving during $[0, t]$ follows readily by observing that

Arrival times are not the same as interarrival times.

$$\mathsf{P}\{N(t) = k\} = \mathsf{P}\{A_k \le t < A_{k+1}\} = \mathsf{P}\{A_k \le t\} - \mathsf{P}\{A_{k+1} \le t\}, \tag{1.2.3}$$

that is, to have precisely k arrivals during $[0, t]$, the arrival time A_k of job k must lie in $[0, t]$, but job $k+1$ must arrive after t. Using the cdf of A_k and partial integration,

$$\begin{aligned}\mathsf{P}\{A_{k+1} \le t\} &= \lambda \int_0^t \frac{(\lambda s)^k}{k!} e^{-\lambda s}\,\mathrm{d}s = \lambda \frac{(\lambda s)^k}{k!} \frac{e^{-\lambda s}}{-\lambda}\Big|_0^t + \lambda \int_0^t \frac{(\lambda s)^{k-1}}{(k-1)!} e^{-\lambda s}\,\mathrm{d}s \\ &= -\frac{(\lambda t)^k}{k!} e^{-\lambda t} + \mathsf{P}\{A_k \le t\}.\end{aligned}$$

Combining this with (1.2.3),

$$\mathsf{P}\{N(t) = k\} = \frac{(\lambda t)^k}{k!} e^{-\lambda t}, \tag{1.2.4}$$

that is, $N(t) \sim \text{Pois}(\lambda t)$, i.e., *Poisson distributed* with parameter λt. It is simple to see from (1.2.4) that

[1.2.8]

$$\mathsf{E}[N(t)] = \lambda t \qquad\qquad \mathsf{V}[N(t)] = \lambda t.$$

Interestingly, we can use (1.2.3) and (1.2.4) to derive the pdf of A_k from the Poisson distribution. From (1.2.4), $\mathsf{P}\{N(t) = 0\} = e^{-\lambda t}$ when $k = 0$, and as $A_0 = 0$ by definition, the RHS of (1.2.3) becomes $1 - \mathsf{P}\{A_1 \le t\}$. Consequently, $\mathsf{P}\{A_1 \le t\} = 1 - e^{-\lambda t}$, and by differentiating this, the pdf is seen to be $f_{A_1}(t) = \lambda e^{-\lambda t}$. For $k \ge 1$, take the derivative with respect to t of (1.2.3) to see that $\mathrm{d}\mathsf{P}\{N(t) = k\}/\mathrm{d}t = f_{A_k}(t) - f_{A_{k+1}}(t)$. It then follows from (1.2.4) that

RHS := right hand side, LHS := left hand side.

$$\lambda \frac{(\lambda t)^{k-1}}{(k-1)!} e^{-\lambda t} - \lambda \frac{(\lambda t)^{k-1}}{(k-1)!} e^{-\lambda t} = f_{A_k}(t) - f_{A_{k+1}}(t).$$

With recursion, we retrieve (1.2.2).

THE FAMILY of random variables $N_\lambda = \{N(t), t \geq 0\}$ is a *Poisson process* with *arrival rate* λ. Once we have N_λ, we can define $N(s,t] := N(t) - N(s)$ as the number of customers that arrive in a general time period $(s,t]$.

A random process *is a much more complicated mathematical object than a random variable: a process is a (possibly uncountable) set of random variables indexed by time, not just* one *random variable.*

Note that $[0,t]$ *is closed at both ends, but* $(s,t]$ *is open at the left.*

It is important to know that N_λ has *stationary and independent increments*. Stationarity means that the distributions of the number of arrivals are the same for all intervals of equal length, that is, $N(s,t]$ has the same probability distribution as $N(u,v]$ if $t-s = v-u$. Independence means, roughly speaking, that knowing that $N(s,t] = n$, does not help to make any predictions about the value of $N(u,v]$ if the intervals $(s,t]$ and $(u,v]$ do not overlap. (1.2.4) implies a number of useful properties of the Poisson process. Next, with small o notation, if $t \ll 1$,

A function $f(h) = o(h)$ *when* $f(h)/h \to 0$ *as* $h \to 0$*, e.g.,* $x^2 = o(x)$.

$$\mathsf{P}\{N(t) = 0\} = e^{-\lambda t} = 1 - \lambda t + o(t), \tag{1.2.5a}$$

$$\mathsf{P}\{N(t) = 1\} = \lambda t e^{-\lambda t} = \lambda t(1 - \lambda t + o(t)) = \lambda t + o(t), \tag{1.2.5b}$$

$$\mathsf{P}\{N(t) \geq 2\} = 1 - \mathsf{P}\{N(t) = 0\} - \mathsf{P}\{N(t) = 1\} \tag{1.2.5c}$$

$$= 1 - (1 - \lambda t) - \lambda t + o(t) = o(t). \tag{1.2.5d}$$

It is also important to know that the reverse of the above holds, that is, a stochastic process with stationary and independent increments and satisfying (1.2.5) is necessarily a Poisson process.

We refer to the literature on (mathematical) probability theory for further background.
Or some other feature by which you want to distinguish between customers.

WHEN WE MERGE two independent Poisson processes N_λ and N_μ, e.g., adults and children entering a shop, we obtain a new Poisson process $N_{\lambda+\mu}$ with rate $\lambda + \mu$. To see this, we use the above. For $t \ll 1$,

$$\begin{aligned}
\mathsf{P}\{N_{\lambda+\mu}(t) = 0\} &= \mathsf{P}\{N_\lambda(t) = 0\}\,\mathsf{P}\{N_\mu(t) = 0\} \\
&= (1 - \lambda t)(1 - \mu t) + o(t) = 1 - (\lambda + \mu)t + o(t), \\
\mathsf{P}\{N_{\lambda+\mu}(t) = 1\} &= \mathsf{P}\{N_\lambda(t) = 1\}\,\mathsf{P}\{N_\mu(t) = 0\} + \mathsf{P}\{N_\lambda(t) = 0\}\,\mathsf{P}\{N_\mu(t) = 1\} \\
&= \lambda t(1 - \mu t) + (1 - \lambda t)\mu t + o(t) = (\lambda + \mu)t + o(t),
\end{aligned}$$

and the third relation in (1.2.5) follow similarly. Moreover, as N_λ and N_μ have stationary and independent increments, $N_{\lambda+\mu}$ has these properties too.

We can also *split*, or *thin*, a stream into several sub-streams. For instance, consider a stream of people passing by a shop as a Poisson process N_λ, and suppose that a customer decides to enter the shop as a Bernoulli distributed rv B, independent of N_λ, with success probability p and failure probability $q = 1 - p$. Then the process of customers entering the shop is a Poisson process $N_{\lambda p}$ with rate λp, because, for $t \ll 1$, from (1.2.5),

Why are there $o(t)$ *symbols already after the first equality signs?*

$$\begin{aligned}
\mathsf{P}\{N_{\lambda p}(t) = 0\} &= \mathsf{P}\{N_\lambda(t) = 0\} + \mathsf{P}\{N_\lambda(t) = 1\}\,\mathsf{P}\{B = 0\} + o(t) \\
&= 1 - \lambda t + \lambda t q + o(t) = 1 - \lambda p t + o(t), \\
\mathsf{P}\{N_{\lambda p}(t) = 1\} &= \mathsf{P}\{N_\lambda(t) = 1\}\,\mathsf{P}\{B = 1\} + o(t) = \lambda t p + o(t), \\
\mathsf{P}\{N_{\lambda p}(t) \geq 2\} &\leq \mathsf{P}\{N_\lambda(t) \geq 2\} = o(t),
\end{aligned}$$

and $N_{\lambda p}$ has stationary and independent increments because N_λ is a Poisson process.

A FINAL INTERESTING, and useful, property of the Poisson process is the distribution of the arrivals over an interal $[0,t]$ given that $N(t) = n$, say. In fact, if this event is true, the arrivals are Unif$[0,t]$.

We write Unif$A$ *for the uniform distribution on the set* A*. This set* A *can be a set of numbers like* $\{0,2,3\}$ *but also an interval like* $[0,t]$*, as is the case here.*

Theorem 1.2.3. Conditional on the event $N(t) = n$, $t > 0$, the distribution of the arrival times $\{A_i\}_{i=1}^n$ is the same as the distribution of the order statistic of n iid rvs $\sim$ Unif$[0, t]$.

Proof. We write for convience $\mathrm{d}t_i = [t_i, t_i + \mathrm{d}t]$, and similarly $\mathrm{d}(t_i - t_{i-1}) = [t_i - t_{i-1}, t_i - t_{i-1} + \mathrm{d}t]$. This notation allows us to write $\mathsf{P}\{X_i \in \mathrm{d}t_i\} = \lambda e^{-\lambda t_i}\,\mathrm{d}t$ when $X_i \sim \mathrm{Exp}(\lambda)$, and $\mathsf{P}\{U_i \in \mathrm{d}t_i\} = \mathrm{d}t/t$ when $U_i \sim \mathrm{Unif}[0, t]$. We provide the proof for $N(t) = 2$; the proof for the general case is nearly identical, but needs more notation.

Consider first the Poisson process with $0 \le t_1 < t_2 \le t$. Then,

$$\mathsf{P}\{A_1 \in \mathrm{d}t_1, A_2 \in \mathrm{d}t_2 | N(t) = 2\} = \frac{\mathsf{P}\{A_1 \in \mathrm{d}t_1, A_2 \in \mathrm{d}t_2, A_3 > t\}}{\mathsf{P}\{N(t) = 2\}}, \qquad (\star)$$

because $N(t) = 2 \iff A_2 \le t < A_3$. For the numerator,

$$\begin{aligned}\mathsf{P}\{A_1 \in \mathrm{d}t_1, A_2 \in \mathrm{d}t_2, A_3 > t\} &\overset{1}{=} \mathsf{P}\{X_1 \in \mathrm{d}t_1, X_2 \in \mathrm{d}(t_2 - t_1), X_3 > t - t_2\} \\ &\overset{2}{=} \lambda e^{-\lambda t_1}\,\mathrm{d}t \lambda e^{-\lambda(t_2 - t_1)}\,\mathrm{d}t e^{-\lambda(t - t_2)} \\ &\overset{3}{=} \lambda^2 e^{-\lambda t}\,\mathrm{d}t^2,\end{aligned}$$

where 1 follows from the recursive relation $A_i = A_{i-1} + X_i$; 2 because the X_i are iid $\sim$ Exp(λ); 3 is algebra. Because $\mathsf{P}\{N(t) = 2\} = (\lambda t)^2 e^{-\lambda t}/2!$, we find that $\mathsf{P}\{A_1 \in \mathrm{d}t_1, A_2 \in \mathrm{d}t_2 | N(t) = 2\} = 2!\,\mathrm{d}t^2/t^2$.

As for the iid uniform rvs, $\mathsf{P}\{U_1 \in \mathrm{d}t_1, U_2 \in \mathrm{d}t_2\} = \mathrm{d}t^2/t^2$. However, there are 2! ways that the uniform rvs hit the set $\mathrm{d}t_1\,\mathrm{d}t_2$, hence the probability that the *order statistic* of U_1, U_2 lies in this set is $2!\,\mathrm{d}t^2/t^2$. This equals the conditional probablity in $(\star)$. □

SUMMARIZING, THE POISSON process N_λ and the exponential distribution are very closely related: a counting process $\{N_\lambda(t)\}$ is a *Poisson process* with rate λ if and only if the inter-arrival times $\{X_k\}$ are iid and $X_k \sim \mathrm{Exp}(\lambda)$. Thus, if you find it reasonable to model inter-arrival times as memoryless, then the number of arrivals in an interval is necessarily Poisson distributed. And, if you find it reasonable that the occurrence of an event in a small time interval is constant over time and independent from one interval to another, then the arrival process is Poisson, and the inter-arrival times are exponential.

TF 1.2.1. Let N be a Poisson process with rate λ, and $0 < s < t$. Claim:

$$\{N(0, s] + N(s, t] = 1\} \cap \{N(0, t] = 1\} = \{N(0, s] = 1\}.$$

TF 1.2.2. Assume that $N_a(t) \sim \mathrm{Pois}(\lambda t)$, $N_s(t) \sim \mathrm{Pois}(\mu t)$ and independent. Claim:

$$\mathsf{P}\{N_a(t) + N_s(t) = n\} = e^{-(\mu+\lambda)t} \sum_{i=0}^{n} \frac{(\mu t)^{n-i}}{(n-i)!} \frac{(\lambda t)^i}{i!}.$$

TF 1.2.3. Claim: this program prints 6.

```
X = [0, 1, 2, 3, 4, 7]
A = [1] * len(X)

```

```
for i in range(2, 5):
    A[i] += A[i - 1] + X[i]

print(A[3])
```

TF 1.2.4. The interarrival times $\{X_k\}$ are iid and exponentially distributed with mean $1/\lambda$, and $A_k = \sum_{i=1}^{k} X_i$, $A_0 = 0$. Claim,

$$\mathsf{P}\{A_{k+1} \le t\} = -\frac{(\lambda t)^k}{k!} e^{-\lambda t} + \mathsf{P}\{A_k \le t\}.$$

TF 1.2.5. Claim: this program prints 31.

```
tot, thres = 8, 30
while tot < thres:
    tot += 5

print(tot)
```

THE EXERCISES REQUIRE pen and paper.

Ex 1.2.6. Show that when $X \sim \mathrm{Exp}(\lambda)$ its moment-generating function (MGF) is $M_X(t) := \mathsf{E}\left[e^{tX}\right] = \lambda/(\lambda - t)$, and use this to compute $\mathsf{E}[X]$ and $\mathsf{V}[X]$.

Ex 1.2.7. Use MGFs to prove that A_i has density $f_{A_i}(t) = \lambda e^{-\lambda t} \frac{(\lambda t)^{i-1}}{(i-1)!}$.

Ex 1.2.8. When $N \sim \mathrm{Pois}(\lambda)$ and $\alpha > 0$, show that $\mathsf{E}\left[\alpha^N\right] = e^{\lambda(\alpha-1)}$. Use this to see that $M_{N(t)}(s) = \exp(\lambda t(e^s - 1))$. Then find $\mathsf{E}[N(t)]$ and $\mathsf{V}[N(t)]$.

Ex 1.2.9. If $X \sim \mathrm{Exp}(\lambda), S \sim \mathrm{Exp}(\mu)$ and independent, show that $Z = \min\{X, S\} \sim \mathrm{Exp}(\lambda + \mu)$, hence $\mathsf{E}[Z] = (\lambda + \mu)^{-1}$.

Ex 1.2.10. If the Poisson arrival processes N_λ and N_μ are independent, show that

$$\mathsf{P}\left\{N_\lambda(t) = 1 \,\middle|\, N_\lambda(t) + N_\mu(t) = 1\right\} = \frac{\lambda}{\lambda + \mu}.$$

In words, given that a customer arrived in $[0, t]$, the probability that it is of the first type is $\lambda/(\lambda + \mu)$.

Ex 1.2.11. If $X \sim \mathrm{Exp}(\lambda)$, $S \sim \mathrm{Exp}(\mu)$ and independent, show that

$$\mathsf{P}\{X \le S\} = \frac{\lambda}{\lambda + \mu}.$$

IF THE SOMEWHAT heuristic derivations above do not satisfy your sense for rigor, then you should solve the next set of exercises; otherwise you can skip them.

Th 1.2.12. If the Poisson arrival processes N_λ and N_μ are independent, use moment-generating functions to show that $N_\lambda + N_\mu$ is a Poisson process with rate $\lambda + \mu$.

Th 1.2.13. Show with moment-generating functions that thinning the Poisson process N_λ by means of Bernoulli random variables with success probability p results in a Poisson process $N_{\lambda p}$.

1.3 FIGHTING THE EXPONENTIAL DISTRIBUTION IS FUTILE

Perhaps the mathematical arguments presented in the preceding section have not fully convinced you of the universality of the exponential distribution and its counterpart, the Poisson distribution. To enhance your understanding, let's use simulation to see how the exponential distribution emerges as a sort of law of nature.

The simulation is based on an internship of one my master's students with the task to evaluate the scheduling system for patients with inflammatory bowel disease (IBD) at a hospital. The IBD patients can be characterized as regular and urgent. Regular patients required check-ups approximately every six months, whereas urgent patients needed to be seen within one week, i.e., five working days. Each consultation takes 10 minutes. Immediately following a consultation, regular patients scheduled their next visit for approximately six months later. There are about 1000 patients in the system, and a working year consist of 200 working days. Given that each patient is seen biannually, a physician needed to consult with an average of 10 patients daily. The problem was to design a planning procedure that allowed enough flexibility to meet the time constraints for the urgent patients without leading to idle times for the physicians. Of course, new patient intakes occurred, but for ease we assume that the patient population remains constant.

This procedure has been changed recently; however, the ideas of this case apply much more generally.

$2 \times 1000/200 = 10$.

Planning regular patients six months in advance seems to lead to stable arrival times, but this is not true, as there are many practical disturbances. A significant number of patients rescheduled their appointment date due to, e.g., altered holiday plans. There were no-shows, and yet other patients relocated to other cities, some died. However, even without such practical problems and even when each patient individually shows relatively predictable behavior, the simulation results below show that *at the population level* the inter-arrival times are well modeled as exponentially distributed rvs.

In conclusion, whether the planner plans six months in advance or would not plan at all, there will not be much difference in the variability of the number of patients arriving per day. In other words, the idea that making appointments half a year in advance will reduce variability by a large amount is simply not true. The verdict must be clear then: when rules don't have an effect, it is best to drop them altogether. It is simply better to send patients a reminder for a new appointment after five months after the visit, instead of planning six months in advance. Much less replanning actions are needed, and the probability for no-shows decreases.

IN THE NEXT MODEL we focus on just the regular patients; we neglect the urgent patients as this forms a relatively small group. Assume that the hospital has just 50 regular patients. For ease, scale time such that each patient plans the next appointment uniformly on the interval $[0.9, 1.1]$, independent of previous appointments and other patients.

Like this, the inter-arrival times between two consecutive appointments correspond to half a year (100 working days) plus or minus 10 days (observe that $[0.9, 1.1] \times 100 = [90, 110]$).

The question of interest is to estimate the distribution of the inter-arrival times between any two patients *as seen by the planner* of the hospital.

Of course, when there would be just one patient, this is easy, the planner sees $\sim$ Unif[0.9, 1.1]. But what happens for many patients?

We will use simulation to obtain insight into this question. The answer will be given in Fig. 1.3.1. Here we present the code to make this figure; it is necessary to understand the algorithms to interpret the figure. Hence, I expect you to really read the code.

If you don't understand certain commands of python, just consult the web or ChatGPT.

IN PYTHON WE nearly always start with loading a set of modules. Modules contain specific functionality that not every Python program needs, hence these should only be loaded if required. For instance, the `numpy` module offers support for numerical work, while `matplotlib` is a plotting library. Here, and elsewhere, we place comments *above* the code to which it relates. We will not explain commands that are trivial to understand.

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

To make a pdf file with a plot that looks nice in LaTeX we use the `seaborn` library and provide it with some extra options, such as good font sizes.

Such configurations are not algorithmic, so you don't have to memorize this.
While designing a figure, I comment out this code because running LaTeX slows down the script.

```
def initialize_plots():
    import seaborn as sns

    rc = {
        "text.usetex": True,
        "font.family": "fourier",
        "axes.labelsize": 10,
        "font.size": 10,
        "legend.fontsize": 8,
        "xtick.labelsize": 8,
        "ytick.labelsize": 8,
    }
    sns.set(style="whitegrid", rc=rc)


initialize_plots()
```

The simulation environment will contain `num_patients` patients and last for `num_periods` half years. With `default_rng` we can set up a random number generator and by setting a seed we ensure to always get the same random deviates.

Here it is 2, but any number would do.

A random deviate is a sample taken from a distribution; it is not the same as a random variable.

```
num_periods = 10
num_patients = 50
rng = np.random.default_rng(2)
```

We next generate a number of inter-arrival times in the following way. The matrix X has `num_patients` rows and `num_periods` columns. Row k of X corresponds to the inter-arrival times of patient k. Each inter-arrival time is $\sim$ Unif[0.9, 1.1]. With these inter-arrival times, the arrival times for patient k are $A_i^k = A_{i-1}^k + X_i^k$, with $A_0^k = 0$.

The `cumsum` along a row of a matrix adds up elements one by one along a row. As such, the elements of the first column of `A` would be equal to the first column of `X`. To ensure that the first row of `A` is zero, so that all patients start at time 0, we set the first column of `X` to 0. However, by setting the first column of inter-arrivals X to zero, we effectively remove one arrival. Thus, we should generate `num_periods + 1` of inter-arrival times.

Along `axis=1`

The `cumsum` of the sequence of numbers $1,3,5$ is $1,4,9$. Hence, if we want to implement $A_0 = 0$, we set just set $X_0 = 0$ and keep $X_1 = 3, X_2 = 5$, so that `A = [0, 3, 5].cumsum()` expands to $A = [0,3,8]$.

```
X = rng.uniform(low=0.9, high=1.1, size=(num_patients, num_periods + 1))
X[:, 0] = 0  # set first column of X to 0
A = X.cumsum(axis=1)
```

We need an environment to control the plots. The `fig` object can be thought of as a canvas on which we draw the plots, and `axes` is a list of `ax` objects, where each `ax` represents a single plot within the figure. As is apparent from Fig. 1.3.1, we have a matrix of four `ax` objects. For easy reference in the code below, we flatten the `axes` object and unpack it to the four separate objects `ax1`, ..., `ax4`.

Ask ChatGPT for further explanation.

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(6, 3))
ax1, ax2, ax3, ax4 = axes.flatten()
```

Next we build the left upper panel of Fig. 1.3.1. Each row contains a number of dots, each corresponding to an arrival event of a patient. In other words, the coordinate of the ith dot of patient k is (A_i^k, k). As we have 50 patients, we have 50 such rows of dots. The for loop adds the dots of each patient to the upper left panel referenced to by `ax1`. To beautify the plot, we give a few options to the plot command. We don't want the dots to be connected by lines, so we set the linestyle `ls='None'`. The marker style is a dot, the markersize `ms=0.2` and the color is blac'k'.

I often use the abbreviations offered by matplotlib as it saves time.

```
ax1.set_xlim(0, 11)
for i in range(num_patients):
    x = A[i, 1:]
    y = i * np.ones(num_periods)
    ax1.plot(x, y, ls='None', marker="o", ms=0.2, c="k")
```

We plot the arrivals as seen by the planner as a jitterplot, that is, we add small random noise to the data points along the y-axis. By spreading out the points slightly, the jitter plot allows for a clearer visualization of the distribution of individual points. We plot the dots at $y = 0$ plus a jittering distributed as $\sim$ Unif$[-1/2, 1/2]$. Without the jitter we would just see small line segments instead of the somewhat rectangular shapes.

```
for i in range(num_patients):
    x = A[i, 1:]
    y = rng.uniform(-0.5, 0.5, size=num_periods)  # add jitter
    ax1.plot(x, y, ls='None', marker="o", ms=0.2, c="k")
```

OUR NEXT TASK is to estimate the density of inter-arrival times as observed by the planner, and plot this in the right panel of Fig. 1.3.1. Recall that each row of the matrix A contains the arrival times of one specific patient. By flatting A we obtain all arrival times in one long list. By sorting, we obtain the arrival times in order of time. Finally, inter-arrival times for the planner are the times between the elements of this list.

```
A = np.sort(A.flatten())
X = A[1:] - A[:-1]
```

We assemble all inter-arrival times of X in 50 bins. On average, we have $\lambda = 50$ arrivals per interval $[0,1]$; thus, the mean inter-arrival time is $1/\lambda = 1/50$. Next, recall that the pdf of exponential distribution is $\lambda e^{-\lambda x}$. To plot the pdf, we take as support for the plot 5 times the mean interarrival time, i.e, $5/\lambda = 0.02$. We then plot the histogram of the interarrival times and the pdf on `ax2`, which corresponds to the upper right panel. The argumnent `density = True` normalizes the area of the bars in the histogram to one, thereby enabling us to compare the histogram to the plot of the pdf.

```
bins = np.linspace(0, 5 / num_patients, 50)
exp_density = num_patients * np.exp(-num_patients * bins)
ax2.hist(X, bins, density=True, color="k")
ax2.plot(bins, exp_density, ":", c="k", lw=0.75, label="pdf")
ax2.legend()
```

WHAT CAN WE see in upper panels of Fig. 1.3.1? In the left upper panel, the arrival times of each patients are at first tightly concentrated around 1, but as time progresses, the arrival times become more and spread out. The jitter plot around $y = 0$ also demonstrates this. The rectangles become wider until they nearly overlap. In the right upper panel, we see that in comparison to the exponential density there are relatively many short inter-arrival times. This is as expected, because all patients start at the same time, hence the arrival times will be concentrated for the first couple of visits. So, to let all patients start at the same time is not a good model for a hospital that has been serving patients for years.

But even with these unreaslistic starting times, the exponential distribution is not a bad fit.

To repair for these unnatural starting times, we set the initial arrival time of each patient as uniformly distributed on $[0,1]$.

```
X[:, 0] = rng.uniform(low=0, high=1, size=num_patients)
```

With this change, we run the above code again, and plot the results in `ax3` and `ax4` to obtain the lower left and right panels of Fig. 1.3.1. Now we see that the jitter rectangles, i.e, the arrivals as seen by the hospital, overlap right from the start, and the exponential distribution is an excellent fit.

It remains to provide the code that saves the figure. The `tight_layout()` function ensures that the labels of the axes are nicely formatted.

```
fig.tight_layout()
fig.savefig("../figures/IBD_exponential.pdf")
```

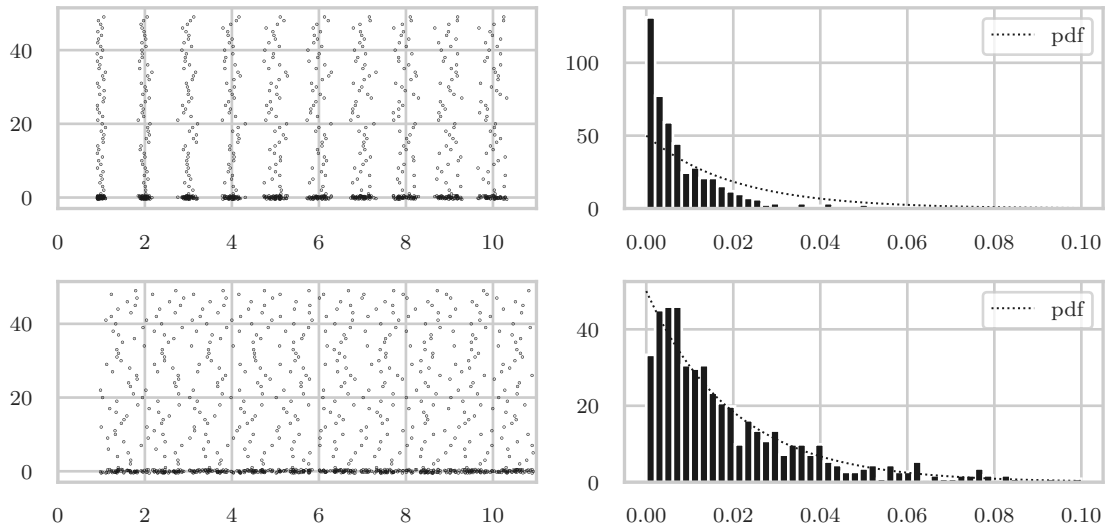


FIG. 1.3.1: *How the uniform distribution leads to the exponential distribution.*

In conclusion, if many patients arrive to the hospital and each adds some small variation to the inter-arrival times, the hospital sees exponential inter-arrival times for the population. Clearly, this is not specific to hospitals: it applies to any system that serves a large pool of customers, each arriving with some variation. To avoid any confusion, here we model the interarrival times for one patient as uniformly distributed, but the same phenomenon occurs when the interarrival times for one patient have other type of distribution.

IT REMAINS TO formalize the observations from the above plots. As a matter of fact, we should distinguish between two limits. Patients recurrently plan appointments, separated around some mean period, e.g., roughly every half a year. The upper left panel in Fig. 1.3.1 suggests that if the simulation duration increases, i.e., the number of appointments made increases, the appointments for each patient becomes uniformly distributed on the period. The lower left panel suggest that if the appointments are uniformly distribued to begin with, they remain uniformly distributed. So, our first step is to prove the claim that in the limit $t \to \infty$, regularly, but still somewhat stochastic, recurring patient arrival times become uniformly distributed on an interval. The second claim is that in the limit of many patients, each arriving at a uniformly distributed time in an interval, the interarrival times of all patients together becomes exponentially distributed, in a scaled sense.

The theorems and proofs are nice and combine order statistics, the beta distribution, first-step analysis, and some infinitesimal calculus to give the reasoning a heuristic flavor. Finally, we discuss an important smoothing property of taking expectations. This idea is of relevancy too in data science when using moving averages. However, if you are not interested in probability theory, skip the rest of the section.

LET US FIRST show that periodically planned arrivals, with some stochasticity in between, tends to become uniformly distributed over an interval. To formalize this claim, we consider a discrete time model: there are on average C days between two successive visits of a patient (for the moment the time on a day is of no importance). For example, take let the iid inter-arrival times X_i be such that $\mathsf{P}\{X_i = C-1\} = \mathsf{P}\{X_i = C+1\} = 1/2$. In this case, if the process starts on day 0 and $X_1 = C-1$, the next visit will be on day $C-1$, but if $X_1 = C+1$, the next visit will be on day 1 of at the next cycle. By looking at the arrival process like this, the patient *moves on a circle* with states $\{0,1,\ldots,C-1\}$, and with every visit the patient moves with probability $1/2$ one state to the left or to the right. Thus, when $X = C-1$, we can just as well model this as $X = -1$.

There is a subtle, small problem when the cycle length is even and $\mathsf{P}\{X = C-1\} = \mathsf{P}\{X = C+1\} = 1/2$. As an even number is a multiple of 2, the odd (even) days can only be visited after an odd (even) number of visits. The problem is easily resolved when assuming, in addition, that C is odd or $\mathsf{P}\{X = 0\} > 0$.

Theorem 1.3.1. Assume the arrival times $\{A_i\}$ live on the circle space $\{0,1,\ldots,C-1\}$, i.e., $A_i = (A_{i-1} + X_i) \bmod C$. If $\{X_k\}_{k=1}^{\infty}$ is a set of iid rvs such that $\mathsf{E}[X_k] < \infty$ and $\sup_j \mathsf{P}\{X_k = j\} < 1$, then $\alpha_k(i) = \mathsf{P}\{A_k = i\}$, i.e., the probability that the kth arrival is on the ith day, converges to $1/C$ as $k \to \infty$.

We impose the condition $\mathsf{E}[X_k] < \infty$ to prevent to having to deal with $A_k = \infty$ for some finite k, and we need $\sup_j \mathsf{P}\{X_k = j\} < 1$ to exclude trivial rvs.

A rv X is trivial when $\mathsf{P}\{X = a\} = 1$ for some a.

Proof. For ease assume for the moment that $\mathsf{P}\{X_k = 1\} = p \in (0,1)$ and $\mathsf{P}\{X_k = -1\} = q = 1-p$, and let C be odd. By first-step analysis,

$$\alpha_{k+1}(i) = p\alpha_k(i-1) + q\alpha_k(i+1).$$

Clearly, if $\alpha_k(i) = 1/C$ for all i, then $\alpha_{k+1}(i) = 1/C$ as well. However, if not all $\alpha_k(\cdot)$ are equal, there must be at least one state i such that $\alpha_k(i)$ is strictly larger than at least one of its neighbors, that is, $\alpha_k(i) \geq \max\{\alpha_k(i-1), \alpha_k(i+1)\}$ and $\alpha_k(i) > \min\{\alpha_k(i-1), \alpha_k(i+1)\}$. But then, on the next visit,

Observe for this argument it is necesseary that C is finite.

$$\alpha_{k+1}(i) = p\alpha_k(i-1) + q\alpha_k(i+1) < \alpha_k(i).$$

In words, as long as not all $\alpha_k(i)$ are equal for all i, after each visit, the maximal α_k becomes lower and, by symmetry, the minimal α_k becomes higher. As the minimum and the maximum cannot cross after any step, all $\alpha_k(i)$ have to converge to the same value as $k \to \infty$. Since there are C states, $\lim_{k\to\infty} \alpha_k(i) \to 1/C$ for all $i \in \{0,1,\ldots,C\}$.

The argument is easy to generalize. Let $\mathsf{P}\{X = j\} = p_j$. By assumption $p_j \in (0,1)$. Then, $\mathsf{E}[\alpha_k(i+X)] = \sum_j p_j \alpha_k(i+j) < \alpha_k(i)$ if $\alpha_k(i) \geq \alpha_k(\cdot)$ when there is at least one j such that $\alpha_k(i+j) < \alpha_k(i)$ and $p_j > 0$. And similarly for the minimum. All in all, this implies that $\alpha_k(i) \to 1/C$ for all i as $k \to \infty$.

Clearly, the proof applies to any finite $C = 1,2,\ldots$. To see how to generalize the argument to the circle $[0,1)$ on the reals, consider a continuous function f on the unit circle C. As the circle is compact and f continuous, f achieves its maximum $f(\bar{x})$ at $\bar{x}$, say. Suppose for some $\epsilon > 0$ there is set $A_\epsilon(\bar{x}) = \{x \in C : f(x) \leq f(\bar{x}) - \epsilon\}$ such that $\mathsf{P}\{A_\epsilon(\bar{x})\} > 0$. (If there is no such set for any ϵ, then f is a constant.) But then, $\mathsf{E}\big[f(x+X)\big] \leq (f(\bar{x}) - \epsilon)\,\mathsf{P}\{A_\epsilon(\bar{x})\} + f(\bar{x})(1 - \mathsf{P}\{A_\epsilon(\bar{x})\}) < f(\bar{x})$. Again, taking an average over f makes the maximum smaller and the minimum larger. □

Applied to the hospital, after many visits of a patient, each planned with about half a year in between, the visits will appear to be uniformly distributed on a 'circle' with a length of half a year.

There is one point that remains about the left lower panel of Fig. 1.3.1; recall that for this panel we started the simulation with uniformly distributed arrival times. What is the distibution of the arrival times *on the circle* if we start like this? Let's analyze this for the case in which the interarrival times are uniform on $[0,1]$ and we take the circle as $C = [0,1)$. Observe that when $C = [0,1)$, $U+V \bmod 1$, i.e., the sum of two iid uniform rvs on the circle, is just the fractional part of $U+V$.

Theorem 1.3.2. Let $Z = \sum_i^n U_i$ for n idd rvs $U_i \sim \text{Unif}[0,1)$. The fractional part of Z, i.e., $Z - \lfloor Z \rfloor$, is uniformly distributed on $[0,1)$.

$\lfloor x \rfloor$ is the integer part of $x \in \mathbb{R}$.

Proof. Realizing that the density of the sum of two uniform rvs on $[0,1]$ is a triangle with base $[0,2]$ and height 1,

$$\begin{aligned}\mathsf{P}\{U+V-\lfloor U+V\rfloor \le x\} &= \mathsf{P}\{U+V\le x\} + \mathsf{P}\{1\le U+V\le 1+x\}\\ &= x^2/2 + (1/2-(1-x)^2/2) = x.\end{aligned}$$

Therefore, the fractional part of the sum of two uniform rvs is uniform on $[0,1]$. We can apply this result from left to right in the summation in Z, thereby proving the claim. □

We now deal with the case with many patients, that is, when the number n of patients goes to infinity. By the above results, we model the arrival times as iid uniform rvs $\{U_k\}_{k=1}^n$ such that $U_k \sim \text{Unif}[0,1)$. Recall that in the simulation we sorted the arrival times of all patients together to obtain the ordered sequence of arrivals as seen by the hospital. In probabilistic terms, the hospital sees *the order statistic* of $\{U_k\}_{i=1}^n$. Write this as $0 = A_0^n < A_1^n < A_2^n < \cdots < A_n^n < A_{n+1}^n = 1$. Define the size of the interarrivals as seen by the hospital as $X_k^n = A_k^n - A_{k-1}^n$, $k = 1,\ldots,n+1$.

We neglect the probability of simultaneous arrivals; these have probability zero.

When the size n of the population increases, the size of the first gap, i.e., the (properly scaled) arrival time of the first patient, will be more and more like an exponentially distributed rv.

We need to scale with a factor n because we have more and more patients.

Lemma 1.3.3.

$$\lim_{n\to\infty} \mathsf{P}\left\{A_1^n \le x/n\right\} = \lim_{n\to\infty} \mathsf{P}\left\{X_1^n \le x/n\right\} = 1 - e^{-x}.$$

Proof. The probability that the smallest of n rvs is less than some x is the same as 1 minus the probability that all n rvs are larger than x. Therefore,

$$\mathsf{P}\left\{A_1^n \le x/n\right\} = 1 - \mathsf{P}\{\min\{U_1,\cdots,U_n\} > x/n\} = 1-(1-x/n)^n,$$

because $\mathsf{P}\{U_i > x/n\} = 1 - x/n$, and the U_i are iid. The RHS converges to $1-e^{-x}$ as $n\to\infty$. Finally, from the definition, $X_1^n = A_1^n - A_0^n = A_1^n$. □

We next show that all interarrival times for the hospital have the same density.

Lemma 1.3.4. For $r \in (0,1)$, the density of the kth interarrival times is $f_{X_k^n}(r) = n(1-r)^{n-1}$, for $k = 1,\ldots n$.

Proof. From the proof of the previous lemma, we have that $\mathsf{P}\{X_1^n \le r/n\} = 1-(1-r)^n$, therefore, $f_{X_1^n}(r)\,\mathrm{d}r = \mathsf{P}\{X_1^n \in [r, r+\mathrm{d}x]\} = n(1-r)^{n-1}\,\mathrm{d}r$. By symmetry, $X_1^n \sim X_{n+1}^n$. It remains to deal with the intermediate interarrivals.

The probability of the event $A_1^n < A_2^n < \cdots < A_{k-2}^n < A_{k-1}^n \in [x, x+\mathrm{d}x] < A_k^n \in [y, y+\mathrm{d}y] < A_{k+1}^n < \cdots A_n^n$ is given

$$f_{A_{k-1}^n, A_k^n}(x,y)\,\mathrm{d}x\,\mathrm{d}y = \frac{n!}{(k-2)!(n-k)!}x^{k-2}(1-y)^{n-k}\,\mathrm{d}x\,\mathrm{d}y,$$

because $k-2$ arrivals must occur before k, $n-k$ after y and one arrival in $[x, x+\mathrm{d}x]$ and one in $[y, y+\mathrm{d}y]$. With this, the probability that $X_k^n \in [r, r+\mathrm{d}r]$ becomes

$$f_{X_k^n}(r)\,\mathrm{d}r = \mathrm{d}r\int_0^{1-r} f_{A_{k-1}^n, A_k^n}(x, x+r)\,\mathrm{d}x,$$

because $X_k^n = A_k^n - A_{k-1}^n = rimpliesy = A_k^n = A_{k-1}^n + r = x + r$, and x can lie anywhere in $[0, 1-r]$.

Substituting in this integral the density $f_{A_{k-1}^n, A_k^n}$, but dropping the factorials for the moment for notational ease, we find

$$\begin{aligned}\int_0^{1-r} x^{k-2}(1-r-x)^{n-k}\,\mathrm{d}x &= (1-r)^{n-1}\int_0^{1-r}\left(\frac{x}{1-r}\right)^{k-2}\left(\frac{1-r-x}{1-r}\right)^{n-k}\\ &= (1-r)^{n-1}\int_0^1 x^{k-2}(1-x)^{n-k}\,\mathrm{d}x\\ &= (1-r)^{n-1}\frac{(k-2)!(n-k)!}{(n-1)!},\end{aligned}$$

where we use the normalization constant of the β distribution to compute the last integral. Canceling the factorials proves the claim. □

Combining the above two lemmas leads straightaway to the next theorem.

Theorem 1.3.5. Let n jobs arrive at a station such that the arrival times are iid rvs $U_k \sim \mathrm{Unif}[0,1]$. Then all interarrival times $\{X_k^n\}_{k=0}^{n+1}$ as seen by the server (the hospital) are iid and $\lim_{n\to\times}\mathsf{P}\{nX_k^n \le x\} = 1-e^{-x}$, i.e, exponentially distributed.

Proof. By Lem. 1.3.3, X_1^n is approximately exponential with parameter $1/n$, but by Lem. 1.3.4, all rvs $\{X_k^n\}_{k=1}^n$ are identically distributed. □

TF 1.3.1. Consider the following chunk of code, where the ith row of X corresponds to the interarrival times of the ith patient to a hospital.

```
import numpy as np

rng = np.random.default_rng(3)
N, T = 5, 8
a, b = 0, 3
X = rng.uniform(low=a, high=b, size=(N, T))
X[:, 0] = 0
A = X.cumsum(axis=1)
```

Claim: the i-th row of A contains the T arrival times of the i-th patient.

TF 1.3.2. Claim: this program prints 24.

```
x = [3 * i + 1 for i in range(10)]
print(x[8])
```

TF 1.3.3. Claim: the keys and the values of the dict a are 8, 13 and 18.

```
a = {}
start, thres = 3, 19
while True:
    start += 5
    if start >= thres:
        break
    a[start] = start

print(a.keys())
```

TF 1.3.4. Claim: this program prints 64.

```
def doit(a):
    match a:
        case int():
            return 8 * a
        case float():
            return 9 * a
        case _:
            raise ValueError("Unknown type passed")

print(doit("8"))
```

TF 1.3.5. Given that the function `uniform()` returns a uniform random number on the interval $(0,1)$ then the following function generates a random variate with a Poisson distribution with parameter $\lambda =$`labda`.

```
def Pois(labda):
    R = 0
    T = - np.log(uniform())
    while T < labda:
        R += 1
        T += -np.log(uniform())
    return R
```

1.4 PROBABILISTIC ARITHMETIC

In these lecture notes we will use quite a lot of programming (in Python) to demonstrate how to obtain numerical insight into the stochastic processes under study. However, rather than being satisfied with a hacky, sloppy, ugly software product, we will develop clean and concise code that matches the language we use to express our ideas to model and analyze stochastic systems. More generally, by focusing on *patterns*, coding becomes a very

interesting intellectual challenge, and using a good *language* helps us prevent making subtle errors. To demonstrate how this process works, we will build a python class to deal with *probabilistic arithmetic*; in the sequel we will use this class numerous times. You should read the code well: your understanding of coding, probability theory and numerical analysis will deepen quite a bit.

At the end we use this code to provide numerical evidence for Th. 1.3.1.

LET US START with making explicit by what we mean by a pattern. As a first pattern, consider the + operator. Suppose we just know how to compute $3+5$, i.e., addition on the positive integers. Observe that definition of addition cannot be applied to rationals, that is, this definition does not suffice to compute $1/2+1/3$. To add fractions, we need to extend the symbol + to the procedure: make the denominators of the fractions identical, then add the numerators. However, this procedure does not work when we want to compute $\sqrt{2}+\sqrt{3}$: $\sqrt{2}$ is not a rational, so we cannot make the denominators equal. Thus, to add general real numbers, we need to extend the definition of + yet further. For this, we first define the number $\sqrt{2}$ properly as $\sup\{x \in \mathbb{Q} : x^2 \leq 2\}$, and with this idea we define $\sqrt{2}+\sqrt{3}$ as $\sup\{x+y : x, y \in \mathbb{Q}, x \leq \sqrt{2}, y \leq \sqrt{3}\}$. Next, we need to define addition for complex numbers, vectors, matrices, ...In summary, you have been *overloading* the + operator from integers to, e.g., fractions, many times before, and similarly for other operators such as minus, product, division, ...

In our present case, we want to *write* $X+Y$ for independent discrete random variables X and Y, and in that case we know from probabilty theory that addition is the *convolution* of X and Y,

$$\mathsf{P}\{X+Y=z\} = \sum_i \sum_j p_i q_j \mathbb{1}_{i+j=z}, \tag{1.4.1}$$

where $p_i = \mathsf{P}\{X=i\}$ and $q_j = \mathsf{P}\{Y=j\}$.The indicator $\mathbb{1}_A = 0$ if the event~A occurs, and~0 otherwise.} In other words, by writing $Z = X+Y$ we mean that we want to obtain the pmf of Z in terms of the pmfs of X and Y. Thus, the first pattern that helps us to reason in more abstract ways about problems is to use *operator overloading*.

{

This brings us to a second pattern: observe that the expression $X+Y$ combines two rvs X and Y and then applies some function. Similarly, the operations $X-Y$, XY, $\max\{X,Y\}$, are all *functions applied to two independent rvs*. What we therefore need is a *method* to compute the pmf of a new random variable $f(X,Y)$, where f is some general *binary* function..

A binary function depends on two arguments

We identify a third pattern when we realize that to compute $\mathsf{V}[X]$, we need $\mathsf{E}[X^2]$. But $\mathsf{E}[X^2]$ is the expectation of the rv $g(X)$ where $g : x \to x^2$. So, we also need a method to compute the pmf of the rv $g(X)$, where now g is some general *unary* function..

A unary function depends on just one argument

In summary, we want to build code that support for rvs:

1. operator overloading for addition, subtraction, and potentially other arithmetical operations,

2. the application of a unary function to obtain a new rv

3. the application of a binary function.

We call these procedures probabilistic arithmetic, and we will build, step by step, elegant and generic code to support this. However, we first need to introduce some computer science concepts.

A USEFUL DATA STRUCTURE to store the pmf of a discrete rv is a dict.

```
X = {1: 1/2, 2: 1/2}
Y = {1: 1/2, 2: 1/2}
```

For these two rvs, it is obvious that $Z = X + Y$ should like `Z = {2: 1/4, 3: 1/2, 4: 1/4}`. Let us use (1.4.1) to compute the pmf of $Z = X + Y$, and see what happens.

```
Z = {}
for i, p in X.items():
    for j, q in Y.items():
        Z[i + j] = p * q
print("The keys: ", Z.keys())
print("The values: ", Z.values())
```

```
The keys:  dict_keys([2, 3, 4])
The values:  dict_values([0.25, 0.25, 0.25])
```

Here is something wrong: the pmf of $Z = 3$ is not $1/2$, as it should. The reason is that `Z[3]` gets overwritten by the = operator in line 4. Actually, we need something that adds probabilities, rather than just setting it, as happens now. The correct datastructure is a `defaultdict(float)` with provides a default value of 0 when a key in the dict is missing.

```
from collections import defaultdict

Z = defaultdict(float)
for i, p in X.items():
    for j, q in Y.items():
        Z[i + j] += p * q # note the +=
print(Z)
```

```
defaultdict(<class 'float'>, {2: 0.25, 3: 0.5, 4: 0.25})
```

Now the result is correct. Note that we use the operator += which is a very convenient way to add up things. To see why it is useful consider the next code. In particular, the third line is a much cleaner (shorter) way to add 3 to the variable name.

```
very_long_variable = 3
very_long_variable = very_long_variable + 3
very_long_variable += 3 # compare this to the line above
```

The above code works in principle, but dealing with the pmf as a dictionary requires some care. Here is an example to understand the problem. I suppose you agree that $1/3 = 1 - 2/3$, so let's see what happens if we run the next code.

```
Z = defaultdict(float)
Z[1/3] = 1
Z[1 - 2 / 3] += 1
print(Z.keys())
```

```
dict_keys([0.3333333333333333, 0.33333333333333337])
```

The dictionary Z suddenly contains two elements rather than just one. Inspecting the last digit of the keys, we see that in one number the last digit is a 7 and in the other it is a 3. So, for a *computer* $1 - 2/3 \neq 1/3$.

This is not something specific to python; the problem lies in the fact that general numbers in $\mathbb{R}$ cannot be represented as floating point numbers.

To get around this, we round the elements of the support to a fixed precision so that we use fractions as keys instead of floats.

```
from fractions import Fraction

a, b = 1 / 3, 1 - 2 / 3
af = Fraction(a).limit_denominator(1000)
bf = Fraction(b).limit_denominator(1000)
print(a == b) # this should be False
print(af == bf)  # this should be  True
```

```
False
True
```

In the code below we will use the, so-called, λ function. Sometimes functions are so short and simple that we simply don't want to bother to give them a name. For such cases, the λ function serves as an *anonymous function*. Note that to apply the function `lambda x:  3 * x` to the number 8, we need to put `lambda x:  x + 3` between round brackets.

```
print((lambda x: 3 * x)(8))
```

```
24
```

Sometims we need to add many rvs, in particular we like to able to code the equivalent of $\sum_{i=1}^{k} X_i$. Let's first see how this works in python for normal numbers.

```
print(sum(x for x in [1, 2, 3]))
```

```
6
```

There is a small catch though. The `sum` needs a starting element. To see why, consider the next code and output; we will discuss it next.

```
print(sum(x for x in "dog"))
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

So, we pass `x` as a string, and then `sum` fails, telling us that it cannot add an `int` and a `str`. But where does the `int` in the error text come from? As it turns out, `sum` starts from a default value 0 and then sums from left to right. In other words, the above summation over the list `[1, 2, 3]` expands to $0+1+2+3$. This clarifies the error: the sum `0 + "dog"` is not well defined. Below we point out when this problem becomes relevant.

A final general concept we will use is *caching*. By caching it is possible to store the result of a function in a dictionary rather than evaluating the function body again on a second call of the function. For simple functions this is not necessary, but when the evaluation of the function requires substantial numerical work, caching decreases computation times beyond imagination (milli-seconds versus more than a century for the same result.)

```
from functools import cache

@cache
def f(x):
    print("The body of the function is called.")
    return 8 * x


print(f(3)) # first call: run function body
print(f(3)) # second call, retrieve from memory, don't run body
print(f(4)) # new argument: run function body
print(f(4)) # second call with argument, don't run body
```

```
The body of the function is called.
24
24
The body of the function is called.
32
32
```

Clearly, the statement in the body function is just printed once, which means that the body of the function is run only once. Thus, `@cache` stores for a given value (here 3 and 4), the output of `f` and does not evaluate the body of `f` for a second time when the function receives the same value for the argument.

WE NOW TURN to building a python class to work in a convenient way with random variables.

We start with loading a number of modules that provides functionality we certainly do not want to build ourselves; this would be too complicated

and too much work. We also load the `typing` module because our code below adds typing information, which means that we specify what type of argument (such as float, or int) is used.[1]

[1] *In simple scripts I tend not to add typing. However in code that I plan to use at other places I find this extra information helpful.*

```
import operator
from collections import defaultdict
from fractions import Fraction
from functools import cache
from typing import Callable, Union

import numpy as np
from numpy.random import default_rng

numeric = Union[int, Fraction, float]
```

The meaning of the next variables will become clear later. The function `toFrac` is a convenience function to hide how to convert a float to a fraction of the desired precision. If we want to change the precision at a later stage, or want to change the conversion, we only have to change the code here.

```
max_denominator = 1_000_000
thres = 1e-16  # Reject probabilities smaller than this.
seed = 3  # For the random number generator.


def toFrac(x: float | int | Fraction):
    """ "Convert x to fraction of specified precision."""
    return Fraction(x).limit_denominator(max_denominator)
```

The RV class, which we build in the coming code blocks, stores the pmf and the support of a rv as a dictionary; for instance, `{0: 1 / 3, 1: 2 /3 }` represents a biased coin with support $\{0,1\}$ and pmf $p_0 = 1/3$ and $p_1 = 2/3$. RV uses `_pmf`, `_support` and `_cdf` as private members.

```
class RV:
    "A random variable whose  keys for the support and values the pmfs."

    def __init__(self, pmf: dict[numeric, float]):
        self._pmf = self.make_pmf(pmf)
        self._support = np.array(sorted(self._pmf.keys()))
        self._cdf = np.cumsum([self._pmf[k] for k in self._support])
```

To make the pmf we use `Fraction` and `defaultdict` for the reasons mentioned earlier. As we do not include outcomes whose probabilities are too small, we need to normalize at the end.

```
    def make_pmf(self, pmf):
        res: dict[Fraction, float] = defaultdict(float)
        for k, pk in pmf.items():
            res[toFrac(k)] += pk if pk >= thres else 0
        res = {k: v for k, v in res.items() if v > 0}  # no prob zero events.
        return self.normalize(res)
```

```
7
8       def normalize(self, pmf):
9           norm = sum(pmf.values())
10          return {k: pk / norm for k, pk in pmf.items()}
```

The next methods provide access to the pmf and the support, and the number of elements in the pmf. When x is not in the support, the pmf is 0.

```
1       def pmf(self, x: numeric) -> float:
2           return self._pmf.get(toFrac(x), 0)
3
4       def support(self) -> np.ndarray:
5           return self._support
6
7       def __len__(self):
8           return len(self._support)
9
10      def __repr__(self):
11          return "".join(f"{k}: {self._pmf[k]}, " for k in self._support)
```

Computing the cdf with binary search via `np.searchsorted` works really fast. With the cdf, the survivor function follows immediately. We don't have to cache the `sf`, because `cdf` already uses caching.

```
1       @cache
2       def cdf(self, x: numeric) -> float:
3           if x < self._support[0]:
4               return 0
5           if x >= self._support[-1]:
6               return 1
7           return self._cdf[np.searchsorted(self._support, x)]
8
9       def sf(self, x: float) -> float:
10          """Survivor function"""
11          return 1 - self.cdf(x)
```

The computation of the mean, variance and standard deviation becomes easy once we realize that all these functions can be expressed in terms of the *pattern* $\mathsf{E}\left[f(X)\right]$ for suitably chosen functions f. In passing, we remark that by taking $f(x) = \mathbb{1}_{x\leq y}$, the cdf $F(x) = \mathsf{E}\left[f(x)\right] = \sum_k \mathbb{1}_{k\leq x} p_k$. However, this is not efficient as compared to the binary search used above. The typing `f: Callable[numeric, numeric]` means the function `f` is supposed to map a number to a number.

```
1       def E(self, f: Callable[[numeric], numeric]) -> float:
2           "Compute E(f(X))"
3           return sum(f(i) * self.pmf(i) for i in self.support())
```

With the expectation defined, the mean, variance and standard deviation follow from simple λ functions. .

```
1       @cache
2       def mean(self) -> float:
```

```
        return self.E(lambda x: x)

    @cache
    def var(self) -> float:
        return self.E(lambda x: x**2) - self.mean() ** 2

    @cache
    def sdv(self) -> float:
        return np.sqrt(self.var())
```

The method `sortedsupport` of `RV` sorts the elements in the support in decreasing order of the probability mass. Below we explain why we need this.

```
    @cache
    def sortedsupport(self) -> np.array:
        "Return the support sorted in decreasing order of the pmf."
        return np.array(sorted(self._pmf, key=self._pmf.get, reverse=True))
```

Additionally, we like to use our `RV` class to generate random deviates that are distributed according to the cdf F. Recall that if $U \sim \text{Unif}[0,1]$, then $F^{-1}(U)$ has the required distribution, because

$$\mathsf{P}\left\{F^{-1}(U) \le x\right\} = \mathsf{P}\left\{U \le F(x)\right\} = F(x),$$

where the last equality follows from the fact that U is uniform on $[0,1]$. Binary sort, i.e., `searchsorted`, is a fast algorithm to compute the inverse $F^{-1}(U)$.

```
    def rvs(self, size: int = 1, random_state=default_rng(seed)) -> np.ndarray:
        "Generate an array with 'size' number of random deviates."
        U: np.ndarray = random_state.uniform(size=size)
        pos: np.ndarray = np.searchsorted(self._cdf, U)
        return self.support()[pos].astype(float)
```

We are nearly finished with the class definition of `RV`, but we need two further elements. To add or subtract two instances of `RV` the class needs to know what to do when writing code like `X + Y` or `X - Y`. The private method `__add__` achieves this by calling `compose_function`, which is defined below to compute $f(X,Y)$ for general f. Subtraction uses `__sub__`, and implements it as `X + (- Y)`. We explain the `convert` function in a minute.

```
    def __add__(self, other: 'RV') -> 'RV':
        other = convert(other)
        return compose_function(operator.add, self, other)

    def __neg__(self):
        return RV({-k: self.pmf(k) for k in self.support()})

    def __sub__(self, other: 'RV') -> 'RV':
        return self + (-other)

```

```
    def __truediv__(self, other: 'RV') -> 'RV':
        other = convert(other)
        return compose_function(operator.truediv, self, other)

    def __mul__(self, other: 'RV') -> 'RV':
        other = convert(other)
        return compose_function(operator.mul, self, other)
```

What happens if we would write `2 + X` to mean that the support of `X` needs to be shifted two steps to the right? Actually, we can catch this in our framework by interpreting the number 2 as a trivial random variable Y, i.e., with all probability mass concentrated on the number 2. So, if we convert ints and floats, by means of a function `convert`, to random variables, we are done. Observe that we already used `convert` in `__add__` and `__sub__`.

We are once again overloading the + operator.

```
def convert(rv):
    "Check and convert to rv if necessary."
    match rv:
        case RV():
            return rv
        case int():
            return RV({rv: 1})  # An int is a shift.
        case float():
            return RV({rv: 1})  # A float is a shift too.
        case _:
            raise ValueError("Unknown type passed as a RV")
```

Next, we want to support the `+=` and `-=` operators, so that we can write `X -= Y`. At later stages we might also want to add many random variables with code like `sum(X for _ in range(5))`. To support this, we have to provide `sum` with an initial element to start the summations; recall the earlier `TypeError`. The next two methods enable this functionality for both cases.

```
    def __radd__(self, other):
        # support sum([rv for i in ...])
        return self.__add__(convert(other))

    def __rsub__(self, other: 'RV') -> 'RV':
        return convert(other).__sub__(self)  # mind the sequence of b - a
```

As a last feature, how to check that two rvs are equal? This is easy: when the pmfs are equal. Note that when overloading the method `__eq__` it is necessary to implement the `__hash__` method too. (Ask ChatGPT why.)

```
    def __eq__(self, other):
        return self._pmf == other._pmf

    def __hash__(self):
        return id(self)
```

This finishes the `RV` class.

THE TWO REMAINING patterns are binary and unary functions. To illustrate, finding the pmf of the sum $Z = X + Y$ of two independent rvs X and Y with pmfs p and q requires convolution like so.

There is a much faster method based on Fast Fourier transforms; we don't discuss that here.

$$\mathsf{P}\{Z = k\} = \sum_i \sum_j \mathbb{1}_{i+j=k} p_i q_j.$$

With general functions it works in precisely the same way. In line 5, the string `defaultdict[Fraction, float]` is typing: it says that the defaultdict `c` map floats to floats. Observe that we break the inner `for` loop when the probability becomes too small. This rule assumes that the order in which the elements in the support pass by are in decreasing order of pmf. The method `sortedsupport` of `RV` ensures that this is the case.

```
1  def compose_function(
2      f: Callable[[numeric, numeric], float], X: RV, Y: RV
3  ) -> RV:
4      "Make the rv f(X, Y) for the independent rvs X and Y."
5      c: defaultdict[Fraction, float] = defaultdict(float)
6      for i in X.sortedsupport():
7          for j in Y.sortedsupport():
8              p = X.pmf(i) * Y.pmf(j)
9              c[f(i, j)] += p
10             if p <= thres:
11                 break
12     return RV(c)
```

Finally, here is the implementation of the application of unary functtions to an `RV`.

```
1  def apply_function(f: Callable[[numeric], numeric], X: RV) -> RV:
2      "Make the rv f(X)"
3      c: defaultdict[Fraction, float] = defaultdict(float)
4      for k in X.support():
5          c[f(k)] += X.pmf(k)
6      return RV(c)
```

AS A MATTER of good conduct, let us include some tests. For professional use, the coverage of the tests is too small, but the point I want to make is that a software project (no matter how simple) is only complete *after* passing the tests, *not* before.

```
1  def tests():
2      U = RV({1: 1})
3      V = RV({2: 1})
4      X = RV({1: 1 / 3, 2: 2 / 3})
5      Y = RV({-1: 1 / 3, -2: 2 / 3})
6
7      assert np.all((U + X).support() == np.array([2, 3]))
8      assert -X == Y
9      assert (U + V).pmf(2) == 0
10     assert (U + V).pmf(3) == 1
11     assert np.isclose(U.var(), 0)
12     assert np.isclose(X.pmf(0.99999999999), 1 / 3)
13     assert np.isclose(X.mean(), 1 / 3 + 2 * 2 / 3)
14     assert np.isclose(X.sf(1), 2 / 3)
```

WITH THE RV CLASS we can study the content of Th. 1.3.1 in numerical terms. More specifically, how fast do the arrival times on the circle converge to the uniform distribution? Recall that the arrival times on the circle are defined like $A_k = A_{k-1} + X_k \bmod C$. Thus, for our numerical study, we want to add rvs and apply the function $x \to x \bmod C$ where C is the length of the cycle

The RV class already supports + for rvs. For the modulo function, the next function applies to ints, floats and rvs.

I like to write (and read) code that resembles the mathematical operations I write on paper.

```
def mod(a, b):
    "Compute a modulo b where a can also be a RV."
    match a:
        case int():
            return a % b
        case float():
            return a % b
        case rv.RV():
            return rv.apply_function(lambda x: x % b, a)
        case _:
            raise ValueError("Unknown type passed to mod")
```

We iteratively compute the arrival time position $\sum_{i=1}^{n} X_i \bmod C$ where the interarrival time is $X_i \sim \text{Unif}\{-1,0,1\}$, and $A_0 = 0$. We track the arrival times that occur with the smallest and the largest probability.

```
step = rv.RV({-1: 1 / 3, 0: 1 / 3, 1: 1 / 3})


def minmax(cycle, num):
    position = rv.RV({0: 1})
    mins, maxs = [], []
    for i in range(num):
        position = mod(position + step, cycle)
        pmf = [position.pmf(k) for k in range(cycle)]
        mins.append(min(pmf))
        maxs.append(max(pmf))
    return mins, maxs
```

This code makes the left panel in Fig. 1.4.1; the right panel is made in the same way. The caption discusses our findings.

```
cycle = 5
num = 30
xx = range(num)
mins, maxs = minmax(cycle, num)
ax1.plot(xx, maxs, "x", ms=2, ls=":", lw=1, label="Maxs")
ax1.plot(xx, mins, "o", ms=2, ls=":", lw=1, label="Mins")
ax1.set_title("Cycle 5")
ax1.set_xlabel("Iteration")
ax1.set_ylabel("Probability")
ax1.legend()
```

TF 1.4.1. Claim: this program prints 40.

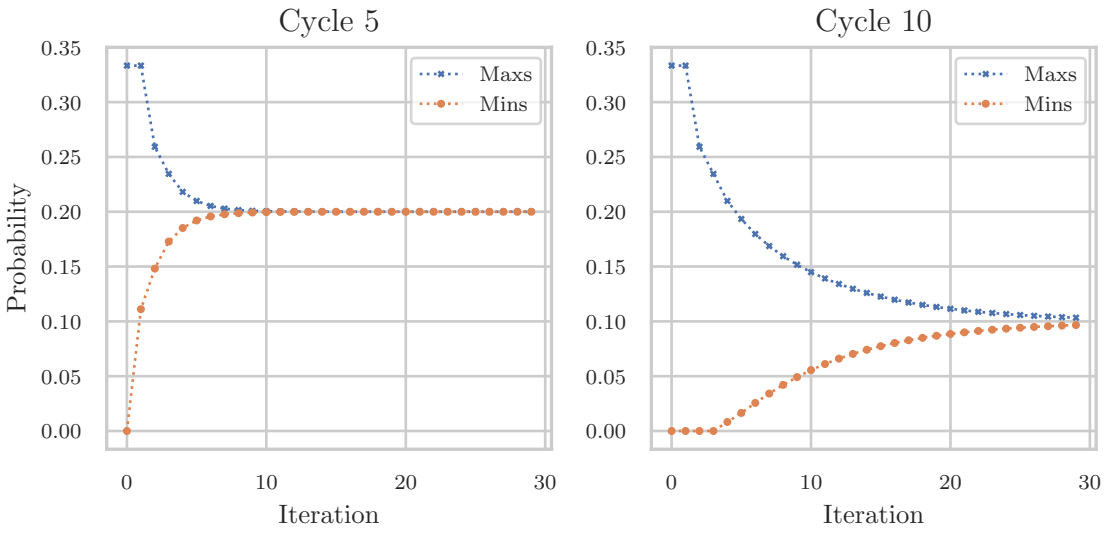


FIG. 1.4.1: *Convergence speed of the time between two visits to the hospital. In the left panel the cycle length $C = 5$, in the right $C = 10$. The largest and the smallest probability on the circle converge to $1/C$ (recall Th. 1.3.1). The larger C, the longer it takes for the sequence of visits to converge to uniform interarrival times, from the perspective of the hospital. In actual practice, this is not a problem, because patients in the hospital start at arbitrary times, not all at time 0.*

```
norm = 5
x = [100, 20, 30, 40]
y = {i: x[i] // norm for i in range(len((x)))}
print(y[3])
```

Ensure you understand the next code really well. Here are some demonstrations to show you how easy it is to make small modifications to test your understanding of probability in general, and coding in particular.

TF 1.4.2. Claim: if we apply this function to two independent rvs X and Y and take `f(i,j) = i/j`, we get a RV that represents $Z = X/Y$.

```
def compose_function(f: Callable[[numeric, numeric], float], X: RV, Y: RV) -> RV:
    """Make the rv f(X, Y) for the independent rvs X and Y."""
    c: defaultdict[float, float] = defaultdict(float)
    for i in X.sortedsupport():
        for j in Y.sortedsupport():
            p = X.pmf(i) * Y.pmf(j)
            c[i, j] += p
            if p <= thres:
                break
    return RV(c)
```

TF 1.4.3. Let the random variable X have cumulative distribution function F_X and let $U \sim \text{Unif}(0,1)$. Claim: $F_X(F_X^{-1}(U)) \sim \text{Unif}(0,1)$.

TF 1.4.4. Let X and Y be two independent rvs with support $\{0,1,2,\ldots\}$ and probability mass functions f_X and f_Y respectively. Claim:

$$\mathbb{P}(X - Y \leq k) = \sum_{i=0}^{\infty} \sum_{i=0}^{\infty} 1\{i - j \leq k\} f_X(i) f_Y(j).$$

Ex 1.4.5. Consult the website of RealPython to improve your understanding of `defaultdict`, the `lambda` function, caching, and classes. If you are interested in somewhat more abstract code, read also about decorators to see how caching is implemented. Of course, you can also ask ChatGPT to explain these concepts: I use it a lot myself and I still learn from the solutions it offers. However, the explanations on RealPython are better and more instructive.

2

CONSTRUCTION OF SIMPLE DISCRETE TIME STOCHASTIC PROCESSES

The first step to analyze stochastic processes is to model it. And for this, there is often not a better start than to build a simulation model for such processes. For this reason, the aim of this chapter is to teach you how to construct and simulate simple queueing and inventory processes.

In Section 2.1 we build discrete-time models of queueing systems, which means that we use the number of jobs that arrive and can be served in fixed periods of time to construct the queueing process. Such a period can be an hour, or a day; in fact, any amount of time that makes sense in the context in which the model will be used. Section 2.3 presents suitable recursions for single-item inventories and demonstrates in particular how to design good rules to control stochastic inventory systems. Once we have some nice models for these two common stochastic processes, we demonstrate in Sections 2.2, 2.4 and 2.5 in detail how to set up simulations in python, compute important performance measures and make graphs of the processes as a function of time.

The length of these periods depends on context. For the present case, it is appropriate to take weeks. For supermarkets perhaps a length of 5 minutes is more appropriate.

2.1 WAITING FOR A PSYCHIATRIST

We start with a case to provide motivation to study queueing systems. Then we develop a set of recursions of fundamental importance by which we can simulate the case and evaluate the efficacy of several policies to improve the system.

AT A MENTAL HEALTH department, five psychiatrists do intakes of future patients to determine the best treatment process once patients 'enter the system'. There are complaints about the time patients have to wait for the intake; the desired waiting time is around two weeks, but the realized waiting time is sometimes more than three months. This is unacceptably long, but . . . what to do about it?

The five psychiatrists put forward various suggestions to reduce the waiting times.

1. Give all psychiatrists the same weekly capacity for doing intakes. Motivation: Currently one psychiatrist does 9 intakes per week, one does 3, and the three other psychiatrists do only 1. This is not a problem during weeks when all psychiatrists are present; however, psychiatrists take holidays, visit conferences, and so on. So, if the psychiatrist with the most intakes per week goes on leave, this affects the intake capacity considerably.

2. Synchronize holidays. Motivation: to reduce the variation in the service capacity, the psychiatrists plan their holidays consecutively. However, perhaps it is better to work at full capacity or not at all.

Later, with the models of Chapter 6, we can immediately see that this will not work, actually they may be detrimental.

3. Control the intake capacity as a function of the waiting time. Motivation: in analogy with supermarkets, scale up (down) capacity when the queue is long (short). Observe that personal often object to such policies because they believe that they have to do more work. However, this is wrong. To see this, suppose that 1000 patients per year need treatment. This number does not depend on whether the patients spend time in queue or not.

We next develop a method to simulate the behavior of the system over time so that we can evaluate the effect of the above suggestions. We use simulation, because the mathematical analysis is too hard.

In case you doubt this, try to analyze transient multiserver queueing systems with vacations, of which this is an example.

LET US START with discussing the essentials of simulating a queueing system. The easiest way to construct queueing processes is to 'chop up' time in periods and develop recursions for the behavior of the queue from period to period. Using fixed-sized periods has the advantage that we do not have to specify inter-arrival times or service times of individual customers; only the number of arrivals in a period and the number of potential services are relevant.

We define:

$a_k =$ the number of jobs that arrive *in* period k,

$c_k =$ the capacity, i.e., the maximal number of jobs that can be served, during period k,

$d_k =$ the number of jobs that depart just before the *end* of period k,

$L_k =$ the *system length*, i.e., the number of jobs in the system at the *end* of period k.

In the sequel we also call a_k the *size of the batch* arriving in period k. Note that the definition of a_k is a bit subtle: we may assume that the arriving jobs arrive either at the start or at the end of the period. In the first case, the jobs can be served in period k, in the latter case, they *cannot* be served in period k.

Since L_{k-1} is the system length at the *end* of period $k-1$, it must also be the number of customers at the *start* of period k. Assuming that jobs arriving in period k cannot be served in period k, the number of customers that depart in period k is therefore

$$d_k = \min\{L_{k-1}, c_k\}, \tag{2.1.1a}$$

Now that we know the number of departures, the number at the end of period k is given by

$$L_k = L_{k-1} - d_k + a_k. \tag{2.1.1b}$$

Like this, if we are given L_0 and numbers $\{a_k\}_{k=1}^n$ and $\{c_k\}_{k=1}^n$, we can obtain the numbers $\{L_k\}_{k=0}^n$ from this recursion.

In a sense, (2.1.1) is the $F = ma$ of a queueing system: Given initial conditions, we apply the rule at time $k-1$ to get the state at time k, and so on.

Of course we are not going to carry out the above computations by hand. Typically, we use company data to model the arrival process $\{a_k\}$ and the capacity $\{c_k\}$, and feed this data into a computer to carry out the recursions (2.1.1). If we do not have sufficient data, we make a probability

model for these data and use the computer to generate random numbers with, hopefully, similar characteristics as the real data. At any rate, from this point on, we assume that it is easy, by means of computers, to obtain random numbers $a_1, \ldots, a_n$ for any n we can reasonably like, and likewise for other sequences of numbers we may need.

CONTINUING WITH THE case of the five psychiatrists, with the above recursions we can now analyze how our proposed rules affect the time patients spend waiting. We mainly want to reduce the long sojourn times.

As a first step in the analysis, we model the arrival process of patients as a Poisson process. The duration of a period is taken to be a week. The average number of arrivals per period, based on data of the company, was slightly less than 12 per week; in the simulation we set it to $\lambda = 11.8$ per week.

NEXT, WE MODEL the capacity in the form of a matrix such that row i corresponds to the weekly capacity of psychiatrist i:

$$\begin{pmatrix} 1 & 1 & 1 & \ldots \\ 1 & 1 & 1 & \ldots \\ 1 & 1 & 1 & \ldots \\ 3 & 3 & 3 & \ldots \\ 9 & 9 & 9 & \ldots \end{pmatrix}. \tag{2.1.2}$$

Thus, psychiatrists 1, 2, and 3 do just one intake per week, the fourth does 3, and the fifth does 9 intakes per week. The sum over column k is the total service capacity for week k of all psychiatrists together.

With such a matrix it is simple to make other capacity schemes. A more balanced scheme would be like this:

$$\begin{pmatrix} 2 & 2 & 2 & \ldots \\ 2 & 2 & 2 & \ldots \\ 3 & 3 & 3 & \ldots \\ 4 & 4 & 4 & \ldots \\ 4 & 4 & 4 & \ldots \end{pmatrix}. \tag{2.1.3}$$

We include the effects of holidays by setting the capacity of one or some psychiatrists to 0 in a certain week. Let us assume that every week just one psychiatrist is on leave such that the psychiatrists' holiday schemes rotate. To model this, we set the entries of the matrix as $C_{1,1} = C_{2,2} = \cdots = C_{1,6} = C_{2,7} = \cdots = 0$, i.e.,

$$C = \begin{pmatrix} 0 & 2 & 2 & 2 & 2 & 0 & \ldots \\ 2 & 0 & 2 & 2 & 2 & 2 & \ldots \\ 3 & 3 & 0 & 3 & 3 & 3 & \ldots \\ 4 & 4 & 4 & 0 & 4 & 4 & \ldots \\ 4 & 4 & 4 & 4 & 0 & 4 & \ldots \end{pmatrix}. \tag{2.1.4}$$

Hence, the total average capacity is $4/5 \cdot (2+2+3+4+4) = 12$ patients per week. There is another simple holiday scheme: when all psychiatrists take

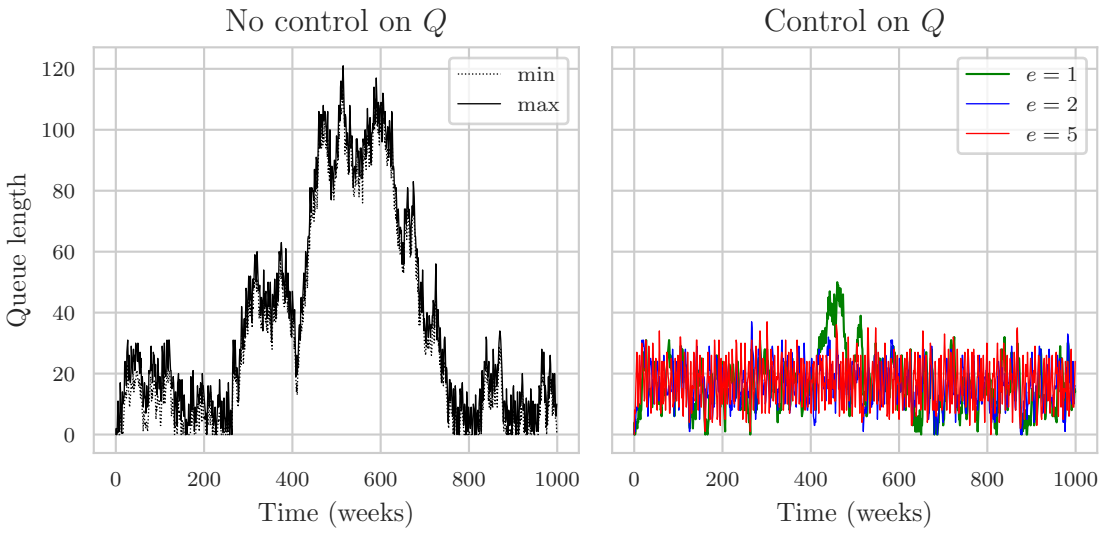


FIG. 2.1.1: *Effect of capacity and holiday plans on the queue length L. The left panel shows the maximum and the minimum queue length under the different holiday and balancing policies. The right panel shows the queue length under the control rule* (2.1.6).

holiday in the same week–corresponds we set entire columns to zero, i.e., $C_{i,5} = C_{i,10} = \cdots = 0$ for week 5, 10:

$$C = \begin{pmatrix} 2 & 2 & 2 & 2 & 0 & 2 & 2 & \dots \\ 2 & 2 & 2 & 2 & 0 & 2 & 2 & \dots \\ 3 & 3 & 3 & 3 & 0 & 3 & 3 & \dots \\ 4 & 4 & 4 & 4 & 0 & 4 & 4 & \dots \\ 4 & 4 & 4 & 4 & 0 & 0 & 4 & \dots \end{pmatrix}. \tag{2.1.5}$$

Of course, random schemes are more likely, but these two are perhaps the simplest that result in the same average capacity. Note that we also apply these the unbalanced capacity plans to obtain four different possibilies.

With these models for the arrivals and the capacities, we use the recursions (2.1.1) to simulate the queue length process for the four different scenarios proposed by the psychiatrists: unbalanced versus balanced capacity, and spread out holidays versus simultaneous holidays.

The results are shown in Fig. 2.1.1. We plot, for each period, the largest and the smallest queue that occurred under all four capacity plans that result from following the first and second suggestions of the psychiatrists. The graphs make clear that these suggestions hardly affect the behavior of the queue length process.

NOW WE CONSIDER Suggestion 3, which comes down to doing more intakes when it is busy, and fewer when it is quiet. A simple rule is to let the capacity for week k depend on the queue length of the week before, for instance,

$$c_k = \begin{cases} 12 + e, & \text{if } L_{k-1} \geq 24, \\ 12 - e, & \text{if } L_{k-1} \leq 12. \end{cases} \tag{2.1.6}$$

We can take $e = 1$ or 2, or perhaps a larger number; the larger e, the larger the control we exercise. We can of course also adapt the thresholds 12 and 24.

Let's consider three different control levels, $e = 1$, $e = 2$, and $e = 5$; when $e = 5$, each psychiatrist does one extra intake. The results, see the right panel of Fig. 2.1.1, show a striking difference indeed. The queue does not explode anymore; already taking $e = 1$ has a large influence.

This simulation experiment shows that changing holiday plans or spreading the work over multiple servers, i.e., psychiatrists, may not significantly affect the queueing behavior. However, controlling the service rate as a function of the queue length can improve the situation dramatically.

THE ABOVE EXAMPLE hopefully claries how we can use simulation to obtain insight into how a stochastic system behaves and how to control it. We next discuss some general points that need further attention.

In (2.1.1) we assume that jobs that arrive in period k cannot be served in period k. If the situation is such that jobs *can* be served in the same period as they arrive, then (2.1.1) should be changed to [2.1.5]

$$d_k = \min\{L_{k-1} + a_k, c_k\}. \tag{2.1.7}$$

Which of (2.1.1) or (2.1.7) to choose depends on what we need to model; in general, no rule is 'best', and what is 'good' depends essentially on the context. For instance, if we like to be 'on the safe side', then it is perhaps best to use (2.1.1) because with this rule, we overestimate the queue lengths, while with (2.1.7) we underestimate the queue lengths.

Note next that in the computation of d_k we make a fundamental modeling choice: if there are jobs in the system and the server capacity $c_k > 0$, the server will serve jobs. There is, however, no formal need to serve jobs even when there is service capacity available. A simple reason not to serve jobs is when it is too expensive, for example, if there is very little demand for a flight on some day then the flight can be canceled. In the models we consider, we will *not* allow to let jobs wait when there is capacity available; in other words, our service processes are assumed to be *work-conserving*.

The above recursions obviously only construct $\{L_k\}$, i.e., the dynamics of the number of jobs in the system. If we also need information about the *sojourn times*, i.e., the time jobs spend in the system, it is necessary to specify the *service discipline*, i.e., a scheduling rule that decides on the order in which jobs in queue are taken into service. In this book we assume henceforth that jobs move to the server in the order in which they arrive. This is known as *First-In-First-Out (FIFO)*. There are many other scheduling rules, such as Last-In-First-Out; we do not discuss these here.

First-Come-First Serve (FCFS) *is also often used.*

It is quite remarkable that the computation of the system length process $\{L_k\}$ is very simple with (2.1.1), but it is much harder to compute the sojourn time J_k for the jobs arriving in period k. In fact, the best we can obtain are bounds on the sojourn time J_k such that $J_k^- \le J_k \le J_k^+$, where

$$J_k^- = \min\left\{m : \sum_{i=k}^{k+m} c_i > L_{k-1}\right\}, \quad J_k^+ = \min\left\{m : \sum_{i=k}^{k+m} c_i \ge L_{k-1} + a_k\right\}.$$

To see how this works, suppose $L_{k-1} = 20$, $a_k = 6$, and $c_k = c = 2$ for all k. Any job that arrives in the kth period, must wait at least $L_{k-1}/c = 20/2 = 10$ periods just to get access to the server under FIFO rule, and we can be sure that the last of these jobs is served after $(20 + 6)/2 = 13$ periods.

We note that there is a difference between *waiting time* W and sojourn time J. The former is the time jobs spend in queue, the latter the time in the system, which is the waiting plus the time at the server L_s. Relatedly, in the model (2.1.1) there is not a job in service, we only count the jobs in the system at the end of a period. Thus, in this model the number of jobs in the system and in queue coincide.

Finally, once the simulation is finished, we compute some performance measures to analyze how the (simulation of the) queueing system behaved. Besides making graphs, we can be interested in the arrival rate, the capacity (also known as the service rate) and mean number in the system,

$$a = \frac{1}{n}\sum_{k=1}^{n} a_k \qquad c = \frac{1}{n}\sum_{k=1}^{n} c_k \qquad L = \frac{1}{n}\sum_{k=1}^{n} L_k.$$

Similary, we can take averages over d_k, J_k^- and J_k^+. The variances of these (simulated) rvs can also be of interest.

IN GENERAL, WITH recursions like (2.1.1) we can carry out numerous what-if-analyses. As another example, a hospital considers to buy a second MRI scanner, because the current one is saturated, as it is used 10 hours per day (from 8 am to 6 pm), and can certainly not serve the forecasted demand. However, if the hospital can identify some extra capacity, it might postpone the purchasing of an additional MRI scanner for a few years. Suppose that a percentage of staff is willing to work from 8 pm to 11 pm, say, and that 30% of the patients is prepared to come to the hospital for a scan between 8 pm and 11 pm, then the capacity would increase with about 30% = (3/10). To see whether this idea is interesting, we can use (2.1.1) to let the computer make a graph of the influence on L. Thus, the recursions such as (2.1.1) and control rules such as (2.1.6) are the essential elements of any queueing model. Once we have the recursions, the computer can compute the performance measures, and then we (humans) have to interpret the results.

Perhaps some staff and patients even prefer to work/have scans in the evening.

An intellectually challenging task sometimes.

Such recursion are not just useful to model queueing systems. For instance, a common model in population dynamics has this form

$$N_{t+1} = N_t + B_t + I_t - D_t - E_t,$$

where N_t is the population size at year t, B_t the number of births, I_t the immigration, D_t the deaths, and E_t the emmigation.

Yet another setting is the reflection of infra-red light by carbon-dioxide. The atmosphere is modeled as a sequence of layers, such that per layer the temperature, air density and chemical composition are considered constant. These properties determine how much infrared light, which is initially reflected by the surface of the earth, makes it to outer space. In each of the layers we need to specify how many infrared photons are transmitted to a higher layer, absorbed, or reflected again to a lower level. Thus, people model a physical process as a queueing system in which photons (customers) go from one layer (station) to another layer (station), or are absorbed (leave the network).

https://github.com/scienceetonnante/RadiativeForcing/blob/main/RadiativeForcing.py

GIVEN THAT FORMULATING recursions is the key step in making useful models for data science, you should practice with this a lot. Here are a large number of exercises to help you make a start with this extremely useful skill.

TF 2.1.1. In a discrete-time queueing system, when job arrivals in period k cannot be served in period k, then $d_k = \min\{L_{k-1}, c_k\}$, $L_k = L_{k-1} - d_k + a_k$.

TF 2.1.2. We have a queueing system in discrete time. Take $c_k = c\,\mathbb{1}_{L_{k-1}>\alpha}$ as service capacity in period k, with $\alpha > 0$. Claim: if $c < \alpha$, and $L_0 > 0$, then $L_k > 0$ for all k.

TF 2.1.3. A single machine serves two queues such that jobs in the first queue get priority over jobs in the other queue. Jobs cannot be served in the period they arrive. Claim: these recursions model this situation correctly:

```
d1[k] = min(L1[k - 1], c[k])
L1[k] = L1[k - 1] - d1[k] + a1[k]
c2[k] = min(L2[k - 1], c[k] - d1[k])
d2[k] = min(L2[k - 1], c2[k])
L2[k] = L2[k - 1] - d2[k] + a2[k]
```

TF 2.1.4. Consider a single-server that serves two parallel queues. Queue i has minimal guaranteed service capacity r^i each period, such that $c_k \geq r^1 + r^2$. Extra capacity beyond the reserved capacity is given to queue 1 with priority. Ariving jobs cannot be served in the period they arrive. (An example is a psychiatrist who reserves capacity for different patient groups.) Claim: these recursions model this situation correctly:

```
c2[k] = min(L2[k - 1], r2)
d1[k] = min(L1[k - 1], c[k] - c2[k])
L1[k] = L1[k - 1] - d1[k] + a1[k]
d2[k] = min(L2[k - 1], c[k] - d1[k])
L2[k] = L2[k - 1] - d2[k] + a2[k]
```

Ex 2.1.5. Prove that the recursion $L_k = [L_{k-1} + a_k - c_k]^+$ generates a system in which jobs can be served in the period they arrive.

2.2 SIMULATING THE PSYCHIATRISTS CASE

In the previous section we showed how to analyze a case in five psychiatrists act as five parallel servers that process a queue of potential patients waiting for an intake. Moreover, we suggested a good rule to control the queue length. In this section we provide the python code we used for the simulation that leads to Fig. 2.1.1.

WE NEED THE next modules.

When discussing code, we will (nearly) always explain the main ideas in the text above each a code block.

```
import numpy as np
from numpy.random import default_rng
import matplotlib.pyplot as plt

```

```
thres_low = 12
thres_high = 24
```

Recall that we compute the queue length recursively by first computing the number of departures $d = \min\{L_{k-1} + a_k, c_k\}$ and then $L_k = L_{k-1} + a_k - d$. The next function follows this logic. It first forms an empty array `L` to store all the queue lengths. Then it applies the recursion by means of a for loop. The loop has to start at $k = 1$ because it 'looks back' at L_{k-1}. For generality, we can provide an optional queue level `L0` if we want the queue to start at another level than 0.

```
def compute_L(a, c, L0=0):
    # L0 is the initial queue length
    L = np.empty(len(a))
    L[0] = L0
    for k in range(1, len(a)):
        d = min(L[k - 1] + a[k], c[k])
        L[k] = L[k - 1] + a[k] - d
    return L
```

Now we make the capacity schemes. The first step is to form an array with 5 rows such that each row corresponds to the weekly capacities of each psychiatrist. Since python starts numbering at 0, instead of 1, the row with index 0 corresponds to the first psychiatrist. We can set all values of the entire row with index 0 with notation like `p[0, :]` where the semicolon refers all indices of the columns. Next, `np.ones(n)` makes a row array of just ones. For each psychiatrist, we fill the related row in `p` with the appropriate capacity of that specific psychiatrist. The result is a matrix that looks like the one in (2.1.2).

It evokes the dots in mathematical notation like the dots here: $i = 1, 2, \ldots$

```
def unbalanced_load(n):
    p = np.empty([5, n])
    p[0, :] = 1 * np.ones(n)
    p[1, :] = 1 * np.ones(n)
    p[2, :] = 1 * np.ones(n)
    p[3, :] = 3 * np.ones(n)
    p[4, :] = 9 * np.ones(n)
    return p
```

The next function builds the matrix (2.1.3)

```
def balanced_load(n):
    p = np.empty([5, n])
    p[0, :] = 2 * np.ones(n)
    p[1, :] = 2 * np.ones(n)
    p[2, :] = 3 * np.ones(n)
    p[3, :] = 4 * np.ones(n)
    p[4, :] = 4 * np.ones(n)
    return p
```

Now we change a capacity matrix such that it includes a certain holiday pattern. This function spreads the holidays over week so that we

obtain (2.1.4). The operator % computes the remainder after division, for example $17\%5 = 2$. In the for loop, the variable j moves one column per iteration, and it cycles over the rows from 0 to 4. The `shape[0]` returns the number of columns of a matrix.

The function does not really need to return p because the change is in-place. *(Ask ChatGPT what is meant with this.) Here we return it so that all functions work in the same way.*

```
88  def spread_holidays(p):
89      n_cols = p.shape[1]
90      for j in range(n_cols):
91          p[j % 5, j] = 0
92      return p
```

In the case of synchronized holidays, we set all capacities in one column to zero, cf. (2.1.5). The `range` function moves from 0 to the end, in steps of 5, so it gives $0, 5, 10, \ldots$.

```
99  def synchronized_holidays(p):
100     n_cols = p.shape[1]
101     for j in range(0, n_cols, 5):
102         p[:, j] = 0
103     return p
```

NOW ALL IS prepared to start the simulation. We set the seed of the random generator, and simulate `num_weeks` of Poisson distributed weekly arrivals. The matrix `L` will contain the queue lengths, one row for each different scenario. As we have four capacity scenarios, we need four rows.

```
110 rng = default_rng(3)
111 num_weeks = 1000
112 a = rng.poisson(11.8, num_weeks)
113 L = np.zeros((4, len(a)))
```

In the first scenario, the capacity is balanced, and the holiday plans are spread over the weeks. The weekly capacity c_i is equal to the sum over all rows in `p` that correspond to week i. This is established by summing over `axis=0` in `p`. For instance, in (2.1.4), the capacity for the first week is $0+2+3+4+4$. Once we have the weekly capacities stored in `c`, we can use the recursion for L to compute the queue lengths at the end of each week. The other scenarios work likewise.

```
117 p = balanced_load(len(a))
118 p = spread_holidays(p)
119 c = np.sum(p, axis=0)
120 L[0, :] = compute_L(a, c)
121
122 p = balanced_load(len(a))
123 p = synchronized_holidays(p)
124 c = np.sum(p, axis=0)
125 L[1, :] = compute_L(a, c)
126
127 p = unbalanced_load(len(a))
128 p = synchronized_holidays(p)
```

```
c = np.sum(p, axis=0)
L[2, :] = compute_L(a, c)

p = unbalanced_load(len(a))
p = spread_holidays(p)
c = np.sum(p, axis=0)
L[3, :] = compute_L(a, c)
```

The code above suffices to make the left panel of Fig. 2.1.1. For the right panel we need control rule (2.1.6). This works nearly the same as the regular recursion for L, but now we add or subtract an extra capacity `e` to the weekly capacity given in the array c. We use the result of [2.1.5] to update L.

```
def compute_L_with_control(a, c, e, L0=0):
    L = np.empty(len(a))
    L[0] = L0
    for k in range(1, len(a)):
        if L[k - 1] <= thres_low:
            c[k] -= e
        elif L[k - 1] >= thres_high:
            c[k] += e
        L[k] = max(L[k - 1] + a[k] - c[k], 0)
    return L
```

We simulate the effect on L for various control rules. We take $e = 1$, $e = 2$ and $e = 5$. We set the weekly capacity to 12 because this is $15 \times 4/5$; recall that in the previous setting one of the five psychiatrists is one leave.

```
c = 12 * np.ones_like(a)
LL = np.zeros((4, len(a)))
LL[0, :] = compute_L(a, c)
LL[1, :] = compute_L_with_control(a, c, 1)
LL[2, :] = compute_L_with_control(a, c, 2)
LL[3, :] = compute_L_with_control(a, c, 5)
```

FINALLY, WE CAN make a figure. It's easy to find on the web what is `ax1` and the other commands. Note, `L.min(axis=0)` computes the minimum queue length per week, and `L.max(axis=0)` the maximum.

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3), sharey=True)
ax1.set_title("No control on $Q$")
ax1.set_ylabel("Queue length")
ax1.set_xlabel("Time (weeks)")

ax1.plot(L.min(axis=0), ":", label="min", color='k', lw=0.5)
ax1.plot(L.max(axis=0), "-", label="max", color="k", lw=0.5)
ax1.legend()

ax2.set_title("Control on $Q$")
ax2.set_xlabel("Time (weeks)")
ax2.plot(LL[1], label="$e = 1$", color='green', lw=0.8)
ax2.plot(LL[2], label="$e = 2$", color='blue', lw=0.5)
ax2.plot(LL[3], label="$e = 5$", color='red', lw=0.5)
```

```
ax2.legend()

fig.tight_layout()
fig.savefig("../figures/psychiatrists.pdf")
```

FOR REAL LIFE situations the recursions can become quite intricate. A nurse takes blood samples at two departments in a hospital. It takes time to walk from one location to another. The nurse serves all patients in one location, then moves to the other location, serves all patients there, and walks the first location, and so on.

This example is a somewhat simplified case from a bachelor thesis.

The recursions can be found by introducing an extra variable p_k that specifies the production state of the nurse. If $p_k = 0$, the nurse serves the first queue, if $p_k = 1$ the nurse moves from queue 1 to queue 2, if $p_k = 2$ the nurse serves queue 2. Finally, if $p_k = 3$ the nurse moves from queue 2 to queue 1. Thus, the production state cycles from $0 \to 1 \to 2 \to 3 \to 0$. Finally, note that expressions like `(3 <= 4)` evalutate to `True` which in turn become 1 in computations. Thus, `(3 <= 4)` implements the indicator function.

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


rng = np.random.default_rng(3)

num_weeks = 40
a1 = rng.poisson(1, size=num_weeks)
a2 = rng.poisson(3, size=num_weeks)
c1 = rng.poisson(3, size=num_weeks)
c2 = rng.poisson(5, size=num_weeks)

L1 = np.zeros(num_weeks)
L2 = np.zeros(num_weeks)
p = np.zeros(num_weeks, dtype=int)  # production state

for k in range(1, num_weeks):
    p[k] = p[k - 1]
    p[k] += (p[k - 1] == 0) * (L1[k - 1] == 0) + (p[k - 1] == 1)
    p[k] += (p[k - 1] == 2) * (L2[k - 1] == 0) - 3 * (p[k - 1] == 3)
    d1 = min(L1[k - 1] + a1[k], c1[k] * (p[k] == 0))
    d2 = min(L2[k - 1] + a2[k], c2[k] * (p[k] == 2))
    L1[k] = L1[k - 1] + a1[k] - d1
    L2[k] = L2[k - 1] + a2[k] - d2

xx = range(num_weeks)
plt.step(xx, L1, where="pre", label="L1")
plt.step(xx, L2, where="pre", label="L2")
plt.step(xx, p, ":", where="pre", label="P")
plt.legend()
plt.savefig("nurses.pdf")
```

TF 2.2.1. If we find the average queue length too large, then we must increase the average service rate to decrease average queue length.

TF 2.2.2. We consider a discrete-time queueing system with a_k the number of arrivals in period k, and c_k the service capacity. Let

$$d_k = \min\{L_{k-1}, c_k\}, \qquad L_k = L_{k-1} - d_k + a_k.$$

Take $a_k \sim \text{Pois}(2)$, $c_k \sim \text{Pois}(1)$, and $L_0 = 0$. Claim: $\mathsf{P}\{L_{10000} \geq 100\} \leq 1/2$.

TF 2.2.3. Consider a single-server that serves two parallel queues. Queue i receives a minimal service capacity r^i every period. Reserved capacity unused for one queue cannot be used to serve the other queue. Any extra capacity beyond the reserved capacity, i.e, $c_k - r^2$, is given to queue 1 with priority. Claim: these recursions are correct.

```
r1, r2 = 9, 1

L1 = np.zeros(num_weeks)
L2 = np.zeros(num_weeks)
for k in range(1, num_weeks):
    d1 = min(L1[k - 1], c[k] - r2)
    c2 = c[k] - d1
    d2 = min(L2[k - 1], c2)
    L1[k] = L1[k - 1] + a1[k] - d1
    L2[k] = L2[k - 1] + a2[k] - d2
```

An example is the operation room of a hospital. There is a weekly capacity, part of which is reserved for emergencies. It might not be possible to assign this reserved capacity to other patient groups, because it should be available at all times for emergency patients. A result of this is that unused capacity is lost. In practice it may not be as extreme as in the model, but still part of the unused capacity is lost. 'Use it, or lose it' often applies to service capacity.

Ex 2.2.4. A machine can switch on and off. Suppose that the jobs that arrive in period k can be served in that period. If the system length hits N (becomes empty) in period k, the machine switches on (off) in period $k+1$. Make the recursions. Assume the machine is off at $k = 0$.

Ex 2.2.5. In the setting of the previous exercise, suppse it costs K to switch on the machine. There is also a cost β per period when the machine is on, and it costs h per period per customer in the system. Can you make a model to compute the total cost for the first T periods?

2.3 CONTROL OF SINGLE-ITEM INVENTORY SYSTEMS

In this section we analyze, so called, single-item inventory systems under *periodic review*. As a simple example, think of boxes of macaroni on a supermarket shelf. At the end of the period, here a day, the supermarket uses the amount of boxes left to decide whether a replenishment of macaroni is necessary or not. If so, it orders in the evening a batch containing ten, say, macaroni boxes to refill the stock. The batch will be delivered by truck at the start of the next day after which a shelf stocker unpacks the batch and puts all ten macaroni boxes on the shelf.

In general there are many different types of items in inventory, ranging from cheap to expensive, so we need different rules to control inventories. We assume that demand is discrete, stochastic, and items are not perishable so that items unsold in one period carry over to the next period. With these assumptions we can develop a few simple, elegant, formulas by which we can construct, hence simulate, the discrete-time dynamics of the most important inventory control systems. Then we define a number of performance measures that allow us to assess the performance of these rules. In a later chapter, after developing some appropriate further mathematical concepts, we can derive closed form expressions for a number of performance measures.

As such, inventory systems are examples of stochastic processes subject to control.

We use the following notation, the meaning of which we explain below:

$$
\begin{aligned}
Q_k &= \text{order size that arrives at the } \textit{start} \text{ of period } k,\\
D_k &= \text{demand arriving } \textit{during} \text{ period } k,\\
I_k &= \text{inventory level at the } \textit{start} \text{ of period } k,\\
I_k^+ &= \text{inventory on hand at the } \textit{start} \text{ of period } k,\\
I_k^- &= \text{backlogged demand at the } \textit{start} \text{ of period } k.\\
P_k &= \text{inventory position at the } \textit{start} \text{ of period } k,\\
l &= \text{fixed lead time.}
\end{aligned}
$$

Inventory theory uses the word 'level' for many different concepts, which should not be confused. Below we introduce inventory level I, order-up-to level S, reorder level s, and service levels.

Note that we are specific about the *timing* of each of the variables; we distinguish between the start and the end of a period. When refering to a quantity measured at the *end* of a period, we mark it with a prime, for instance I'_k.

THE DYNAMICS OF the inventory can be simply constructed if we are given a set of period demands $\{D_k, t = 1,2,\ldots\}$ and order sizes $\{Q_k, t = 0,1,\ldots\}$, but first we need to address a subtle point. In many inventory systems there is a delay between *issuing* a replenishment order and *receiving* this order: only after receiving the order, the order can be used to replenish on-hand stock and serve demand. The time between the placement of the order and its arrival at the inventory is called the *leadtime* l, and the number of orders under way are called *outstanding* orders. In the sequel we take l to be constant over time and independent of the number of outstanding orders. Then, when placing an order of size Q_{k-l} at the start of period $t-l$ so that it arrives at the start of period t, the *inventory level* follows the rules

$$
I'_{k-1} = I_{k-1} - D_{k-1}, \qquad I_k = I'_{k-1} + Q_{k-l}. \tag{2.3.1}
$$

The sequence is important. The replenishment order arrives at the start of the period so that it is available to serve demand.

In general, the inventory level I_k can be positive and negative. In the former case, $I_k^+ = \max\{I_k, 0\}$ is the *inventory on hand* as the items can be used to meet demand directly from stock. In the latter case, $I_k^- = \max\{-I_k, 0\}$ is the *backlogged* demand, because it is demand (or a customer) that has to wait for the arrival of future replenishment order(s).

Clearly, if we would use the inventory level to determine when and how much the order, we might be too late in placing orders, due to the

leadtime. To compensate for this, we use the *inventory position* which updates according to the rule

$$P'_{k-1} = P_{k-1} - D_{k-1}, \qquad P_k = P'_{k-1} + Q_k. \tag{2.3.2}$$

Thus, the inventory position includes the outstanding orders, hence accounts for items under way to cover future demand. In other words, the difference between the inventory position and the inventory level is that in the former we immediately include all orders 'in the books', but these orders physically arrive l periods later, and only then they can be used to meet demand.

Let us next study a number of rules that use the *inventory position* to compute order sizes; these policies differ only in the decision when to trigger an order and how much to order. Since these rules check the inventory position at the end of a period, we say that the inventory is under periodic review.

THE BASESTOCK POLICY seems to be the simplest control rule: it just replenishes for each item demanded. Specifically, under a basestock policy, the size of the order is given by

$$Q_k = S - P'_{k-1}$$

where S is the *order-up-to* level. By combining this ordering rule with (2.3.2), we see that the $P_k = S$ for all t.

Note that 1: The amount ordered depends on the end-of-period *inventory position, and 2: the inventory control policy uses the inventory position, not the inventory level.*

A basestock policy is often used to control inventories of expensive items in which the order cost K is small relative to the price of the item or can be easily included in the selling price of the item. For instance, when somebody buys a washing machine, the customer is often prepared to pay to have the washing machine delivered and installed in place. Also, customers do not buy multiple washing machines at the same time, hence, any ordering (or setup) costs must be covered by each single demand.

AN (s,S) POLICY is appropriate in situations in which the size of the deliveries can vary and the order cost is significant. The update rule is as follows. When the inventory position is at or below the reorder level s, we place an order up to S, specifically,

$$Q_k = (S - P'_{k-1})\,\mathbb{1}_{P'_{k-1} \le s}.$$

Observe that The basestock policy is a special case of the (s,S) policy: just set $S = s+1$. Sometimes supermarkets use (s,S) policies to control the inventory of packaged meat, like bacon. The inventory levels are tracked real time, and there are daily deliveries. As long as the inventory level is sufficiently high, there is no need to replenish (hence a basestock policy is not useful), and, as the items are perishable, it seems best to only replenish when there are very little items left. Then the shelvers should put the newest items under the older items, so that customers (hopefully) pick the oldest items first.

THE (Q,r) POLICY is useful when the ordering cost is large, or can be easily spread over multiple items. The order quantity is taken as a multiple of

a minimal order quantity Q such that $P(t)$ always lies above the reorder level r and below $r+Q+1$. In a formula,

$$Q_k = Q \left\lceil \frac{r+1-P'_{k-1}}{Q} \right\rceil \mathbb{1}_{P'_{k-1}\le r}.$$

Admittedly, it is a bit strange that the (Q,r) policy is not called the (Q,s) policy, but this is the name that is commonly used.

So, a basestock policy is a (Q,r) policy with $Q=1$ and $r=s$. The (Q,r) policy is mostly used by supermarket supply chains to replenish groceries. The factory packets the items in batches, and transports these batches on pallets to distribution centers. At a distribution center, batches are picked from the pallets and send to supermarkets. Finally, at the supermarket a shelver removes the cellophane or box and places the single items on the shelf. Thus, to reduce ordering costs, the entire supply chain orders and transports batches of items, rather than single units. Moreover, as for supermarkets leadtimes are in the order of a day, the order size Q is typically larger than the average daily demand.

FINALLY, THE (T,S) POLICY specifies to order up to S every T periods. In a formula:

$$Q_k = (S - P'_{k-1})\,\mathbb{1}_{\text{mod}\{k,T\}=0},$$

where mod$\{k,T\}$ is the remainder of k/T, In settings in which it is convenient to supply stores in a fixed order, this policy makes sense. The route length determines the number of periods T between the deliveries. Then, at a visit, the inventory is refilled to level S. An example can be highway service areas whose fuel tanks are refilled during the night.

CLEARLY, WHEN WE have a (simulated) sequence of period demands $D_1, D_2, \ldots$ and we have an inventory control rule, we can run the above recursions to compute how the inventory level and position behave over time. It remains to develop a number of measures to assess how the inventory control rule performs, in particular to see how costs depend on the policy parameters.

FOR COST COMPUTATIONS we need to make a choice when to account for the period cost of inventory and backlog. Here we choose to charge b Euro for each demand in backlog and h per item on hand at the start of the period. If there is a cost K per time we place an order, the average cost up to period n becomes

$$Kq + hI^+ + bI^-,$$

where

$$q = \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{Q_k>0} \qquad I^+ = \frac{1}{n}\sum_{k=1}^{n} I_k^+ \qquad I^- = \frac{1}{n}\sum_{k=1}^{n} I_k^-,$$

which, implicitly, depend on the inventory policy. Clearly, we can search for a good policy by running (many) simulations for various policies and comparing the average costs.

SERVICE LEVELS form a set of measures to quantify the extent to which demand is met. Here we discuss three different such measures. The *ready*

rate is the fraction of periods in which the end-of-period inventory is not negative, i.e.,

$$\alpha := \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{I'_k \geq 0}.$$

The *fill rate* β relates to the fraction of demand satisfied from on-hand stock. Observe first that at period t as much as possible of the demand D_k is satisfied. It seems that this should be equal to $\min\{D_k, I_k\}$, but I_k might be negative. Thus, we should take $\min\{D_k, I_k^+\}$ instead, and with this we define the fill rate as

$$\beta := \frac{\sum_{k=1}^{n} \min\{D_k, I_k^+\}}{\sum_{k=1}^{n} D_k}. \tag{2.3.3}$$

The third measure is the *cycle service level* which is defined as the fraction of cycles in which the inventory was not negative; a cycle is the time between the arrival of two consecutive replenishments. To capture this in a formula, note first that a cycle ends in period t when $Q_{k-l} > 0$ (because then a replenishment arrives). Second, there is still on-hand inventory at the start of period k when $I'_{k-1} \geq 0$. Thus, the number of cycles in which the inventory was non-negative must be $\sum_{k=l+1}^{n} \mathbb{1}_{Q_{k-l}>0}\mathbb{1}_{I'_{k-1}\geq 0}$. The cycle service level then follows from averaging over the total number of cycles $\sum_{k=l+1}^{n} \mathbb{1}_{Q_{k-l}>0}$:

$$\alpha_c = \frac{\sum_{k=l+1}^{n} \mathbb{1}_{Q_{k-l}>0}\mathbb{1}_{I'_{k-1}\geq 0}}{\sum_{k=l+1}^{n} \mathbb{1}_{Q_{k-l}>0}}.$$

The cycle service measure is much used in practice because it is simple to execute: when a replenishment arrives, just register whether there is still on-hand stock. However, it is often not an accurate measure of the demand met, hence not helpful to understand how many customers have been served from on-hand stock.

It's an ideal performance measure for managers: easy to understand, but useless.

SOME INVENTORY CONTROL problems can be formulated as a combination of a linear program and some recursions. Consider a production system that can produce maximally M_k items per week during normal working hours, and maximally N_k items during extra (weekend and evening) hours. Management needs a production plan that specifies for the next T weeks the number of items to be produced per week, assuming that demand must be met from on-hand inventory. For this problem, let, for period k,

D_k = Demand in week k,
S_k = Sales, i.e., number of items sold, in week k,
r_k = Revenue per item sold in week k,
X_k = Number of items produced in week k during normal hours,
Y_k = Number of items produced in week k during extra hours,
c_k = Production cost per item during normal hours,
d_k = Production cost per item during extra hours,
h_k = Holding cost per item, due at the end of week k,
I_k = On hand inventory level at the end of week k.

The decision variables are X_k, Y_k and S_k: we can decide how much to make during normal and evening shifts, and we can decide on how much demand to satisfy. the objective is to maximize the rewards over the production horizon T:

$$\max \sum_{k=1}^{T} (r_k S_k - c_k X_k - d_k Y_k - h_k I_k),$$

and the constraints are (for all k),

$$\begin{aligned} 0 &\le S_k \le D_k, \\ 0 &\le X_k \le M_k, \\ 0 &\le Y_k \le N_k, \\ I_k &= I_{k-1} + X_k + Y_k - S_k. \\ I_k &\ge 0. \end{aligned}$$

It remains to fill in the data and feed this to an optimizer.

INVENTORY AND QUEUEING systems are related throught three types of 'buffer': *inventory*, *capacity*, and *time*. These three related concepts are exceedingly useful in the analysis and improvement of nearly any logistic system as they can traded against each other to satisfy demand. Suppose demand can be backlogged, that is, a company does not have to meet (all) demand from stock, but customers agree that they (sometimes) have to wait for their demand to be satisfied. In this case, the company needs *more* time, but *less* inventory to meet demand, which can reduce overall cost. In the limiting case in which it is impossible to stock the product, e.g., operations in a hospital, or very expensive, all demand is backlogged, thereby reducing the inventory system to a queueing system with finite capacity. The company can still increase capacity, as this typically shortens queueing times. However, increasing capacity comes at a cost. All in all, many businesses struggle with how to organize *capacity* and *inventory* levels such that the *time* limitations as imposed by customers are met.

If you consider to become consultant, then memorize this as you can use it time and again.

Hopp and Spearman [2008] offers very nice additional insights on this material.

TF 2.3.1. Simulating inventory control problems is significantly harder than simulating queueing problems, due to the presence of control policies.

TF 2.3.2. Claim: in a single-item inventory system, the inventory level triggers the orders.

TF 2.3.3. Claim: in a single-item inventory system the inventory level only depends on the lead time l when $l > 0$.

TF 2.3.4. Claim: in a single-item inventory system the inventory position and inventory level coincide when the leadtime $l = 0$.

Ex 2.3.5. In production environments it is important to decide which products have to be make-to-order (MTO) and which make-to-stock (MTS). Why is a queueing system a model for MTO production? What inventory control model would best model a queueing system, and what are the parameters?

Ex 2.3.6. Suppose $h \gg b$, i.e., inventory on hand is much more expensive than demand in backlog, what service level would you like?

Ex 2.3.7. Which variables (period, order quantity) are fixed, which can vary, in each of the inventory policies we discussed above? Which policy should be able to achieve the lowest cost?

Ex 2.3.8. Above we assumed that demand in excess of the stock level is backlogged. Modify the recursions of the basestock model such that it can cope with lost demand. For instance, if right after the start of period k, the inventory level $I_k = 5$, and $D_k = 9$, then 4 items are lost, and 5 are accepted.

2.4 SIMULATING INVENTORY SYSTEMS

In this section we develop a simulation environment to analyze a single-time inventory system controlled by an (s, S)-policy.

THE LIGHTHOUSE COMPANY sells ceiling lamps and considers to use an (s, S) policy to control the inventory since the replenishment cost, $K = 50$ euro is relatively high. The company wants to use simulation to find sound values for s and S.

The details of the case are like this. Ceiling lamps sell at 100 Euro while the buying price is 40 Euro. The monthly holding cost is estimated to be 50% of the buying price. To prevent customers from buying a lamp from a competitor (e.g., Internet sales), the Lighthouse Company offers a reduction of 20% per day of the regular selling price when customers cannot be satisfied from on-hand stock. The pmf f of the daily demand is given by

$$f_0 = 1/6, \quad f_1 = 1/5, \quad f_2 = 1/4, \quad f_3 = 1/8, \quad f_4 = 11/120, \quad f_5 = 1/6.$$

The leadtime $l = 2$ days.

WITH THE CODE below we set up a simulation to estimate the performance measures for one specific (s, S) policy.

Most of the next modules should by now be familiar to you. `icecream` offers nice printing functionality.

```
import numpy as np
from numpy.random import default_rng
from icecream import ic  # simple printing

import random_variable as rv
```

Here are the model parameters. The daily demand is a `RV`. The cost parameters have to be such that the units are right; here we convert to cost per day.

```
demand = rv.RV({1: 1 / 6, 2: 1 / 5, 3: 1 / 4, 4: 1 / 8, 5: 11 / 120, 6: 1 / 6})

L = 2
h = 40 * 0.5 / 30  # daily holding cost
b = 100 * 0.2  # daily backlog cost
K = 50

s, S = 3, 20  # The policy parameters
```

We next compute the inventory positions and inventory levels. We generate `N` random deviates for the demand and pass a random generator to ensure to get the same numbers for every simulation run. For ease we combine (2.3.1) into a single recursion for I_t. The first `for` loop for the inventory levels computes I for the first couple of period, because $t - l < 0$ for $t < l$. In the second loop we should protect against a simple error: it might happen that the number of periods over which we simulate is smaller than l.

```
N = 100
rng = default_rng(3)  # set the seed
D = demand.rvs(size=N, random_state=rng)
P = np.zeros(N)
Q = np.zeros(N)
I = np.zeros(N)
P[0] = I[0] = S

for t in range(1, N):
    Pprime = P[t - 1] - D[t - 1]
    Q[t] = (S - Pprime) * (Pprime <= s)
    P[t] = Pprime + Q[t]

# Mind the leadtime L
for t in range(1, min(L, N)):
    I[t] = I[t - 1] - D[t]
for t in range(min(L, N), N):
    I[t] = I[t - 1] - D[t - 1] + Q[t - L]
```

The last block of code computes the measures. The cycle service level α_c is worth to study. We use slicing, which is a technique much used in python, but also in matlab.

Ask ChatGPT for explanation if you find this line hard.

```
ic(D.mean(), P.mean(), I.mean(), Q.mean())

Iplus = np.maximum(I, 0)
Imin = np.maximum(-I, 0)

ic(Iplus.mean(), Imin.mean())

cost = K * (Q > 0).mean() + h * Iplus.mean() + b * Imin.mean()
ic(cost)

alpha = (I >= 0).mean()
beta = np.minimum(D, Imax).sum() / D.sum()
alpha_c = ((Q[:-L] > 0) * (I[L - 1 : -1] >= 0)).sum() / (Q[:-L] > 0).sum()
ic(alpha, beta, alpha_c)
```

IT REMAINS TO find good policy parameters, i.e., s and S. In principle this is simple: just change s and S over a range of values and see how that affects the performance measures. However, finding the optimal parameters *efficiently* is not easy. To minimize the cost, I have published an article on how to do that (for the interested: the optimality proof and computations depend on martingales, renewal reward theory and optimal stopping). However, I did not study how to minimize average cost under some constraint on one of the service level measures, for instance.

Ex 2.4.1. Which inventory rule does the next code implement? What are the policy parameters?

```
import numpy as np

gen = np.random.default_rng(seed=42)

l = 2
n = 100
Q_size = 10
s = 20
labda = 3
D = gen.poisson(labda, size=n)

Pp = np.zeros_like(D)
Q = np.zeros_like(D)
for t in range(1, len(D)):
    Qp[t] = Q_size * (Pp[t - 1] <= s)
    Pp[t] = Pp[t - 1] + Q[t] - D[t]

Ip = np.zeros_like(D)
for t in range(l, len(D)):
    Ip[t] = Ip[t - 1] + Q[t - l] - D[t]
```

2.5 GENERAL BEHAVIOR OF QUEUEING SYSTEMS

In this section we develop a simulation environment to obtain insight into general behavior of queueing systems and make graphs of the evolution of queue length processes.

THE FIRST POINT of a queueing system we should address is its stability. Intuitively, when work arrives faster than it can be served, on average, the queue must increase as a function of time, while when work arrives slower, the queue must stay limited (most of the time). To check this we show how to make Fig. 2.5.1 which contains the graph of the queue length process for a number of different arrival rates and capacities.

We import the standard modules, cf., Section 2.2. The python function below simulates the queue length process for a given sequence of arrivals $a = (a_0, a_1, \ldots, a_n)$ and service capacities $c = (c_0, c_1, \ldots, c_n)$. We set L_0 to `L0` and start simulating from time 1 onward. Thus, the values a_0 and c_0 remain unused. Note that for `d` we assume that the jobs arriving in period i can be served in that same period.

```
def compute_queue_length(a, c, L0=0):
    """Compute queue length L for  arrivals a,  capacities c
    and starting level L0"""
    L = np.zeros_like(a)
    L[0] = L0
    for i in range(1, len(a)):
        d = min(c[i], L[i - 1] + a[i])
        L[i] = L[i - 1] + a[i] - d
    return L
```

Next, we set the parameters for the simulation.

```
labda, mu = 6, 7  # arrival and service rates
L0 = 40  # starting level of queue
num = 50  # run length
x = np.arange(0, num)  # x values for the plot
```

To get a feel for the variability inherent to stochastic systems, we plot a few different sample paths of the queue length process. Thus, we give 10 different seeds to the random number generator, simulate the queue length for each different seed, and add each sample path to `ax1`. For the simulation, we take a_i uniformly distributed on the integers $\lambda-1, \lambda, \lambda+1$ with $\lambda = 6$, so that 6 jobs arrive on average in a period. As the function `rng.integers(l, h)` generates iid uniformly distributed random deviates on $l, l+1, \ldots, h-1$, we should set $h = \lambda+2$ instead of $h = \lambda+1$. The service capacity is constant and equal to $\mu = \lambda+1 = 7$, so the system must be stable.

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3), sharey=True)
ax1.set_xlabel("time")
ax1.set_ylabel("Queue length")
ax1.set_title("Stable system")

for seed in range(10):
    rng = np.random.default_rng(seed)
    a = rng.integers(labda - 1, labda + 2, size=num)
    c = mu * np.ones_like(a)
    L = compute_queue_length(a, c, L0)
    ax1.plot(x, L, linewidth=0.75)
```

We repeat the simulation for a system in which $\mu = 5$, hence we expect this to be unstable. The last step is to write the figure to a file. Fig. 2.5.1 shows the result of our work.

```
mu = 5
ax2.set_xlabel("time")
ax2.set_title("Unstable system")
for seed in range(10):
    rng = np.random.default_rng(seed)
    a = rng.integers(labda - 1, labda + 2, size=num)
    c = mu * np.ones_like(a)
    L = compute_queue_length(a, c, L0)
    ax2.plot(x, L, linewidth=0.75)


fig.tight_layout()
fig.savefig('../figures/queue-discrete-time-stability.pdf')
```

HOW ABOUT REDUCING the difference between λ and μ? When there is little extra capacity, i.e. μ is just a little larger than λ, it must take longer to drain large fluctuations of the queue length than when μ is quite a bit larger than λ. To see this in quantitative terms, we run a simulation, one with $\mu = 6.5$ and another with $\mu = 6.2$ while keeping $\lambda = 6$, and we compute the mean and variance of the number of jobs in the system.

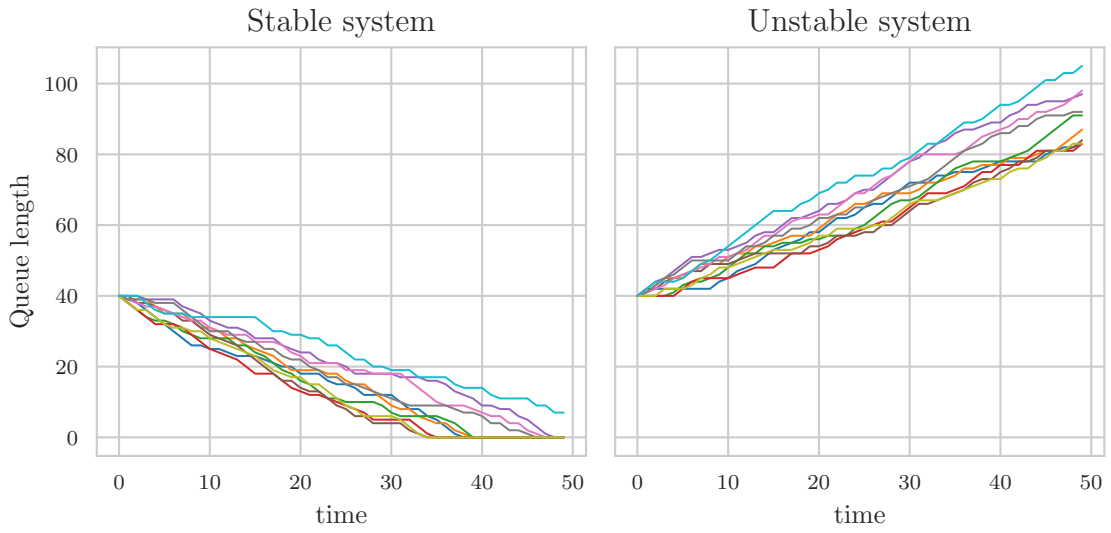


FIG. 2.5.1: *Drift of queue length process. In the left panel, the queue length starts at 40, and the service capacity is $\mu = 7$ while the arrival rate is $\lambda = 6$, so the initial queue drains at about 1 per period. In the right panel, $\mu = 5$, $Q_0 = 40$, and the queue increases at a rate 1 job per period. Clearly, in one case the queue length process decreases, in the other it increases.*

To do so, we need a sequence of random integers for c such that the mean is $\mu = 6.2$, say. One way to make such a sequence is to set $c_i = 6 + b_i$ where b_i is a Bernoulli rv with success probability $p = \mu - 6 = 0.2$.

The code for this is simple because we don't need to make a graph this time. We just add the next code to the code above.

```
78  labda, mu, L0 = 6, 6.1, 0
79  num = 5000
80
81  a = rng.integers(labda - 1, labda + 2, size=num)
82  c = int(mu) * np.ones_like(a)
83  p = mu - int(mu)  # fractional part of mu
84  c += rng.binomial(1, p, size=len(c))
85  print(c.mean())  # just a check
86  L = compute_queue_length(a, c, L0)
87  print(L.mean(), L.var())
```

The results are $\mathsf{E}[L] = 0.7$, $\mathsf{V}[L] = 1.1$ for $\mu = 6.3$, and $\mathsf{E}[L] = 2.2$, $\mathsf{V}[L] = 5$ for $\mu = 6.1$. (Here $\mathsf{E}[L]$ stands for the sample average $n^{-1}\sum_{i=1}^{n} L_i$, and likewise for $\mathsf{V}[L]$.) Indeed, the mean queue length increases when service capacity becomes tighter.

We are in a state of sin here: the mean of a rv is not the same as its sample average. The strong law of large numbers makes a statement only about the limit; for small sample sizes, the difference can be quite large.

Suppose we cannot serve jobs in the period in which they arrive. Then we use the code `d = min(c[i], L[i - 1])` to update `d`. In this case, $\mathsf{E}[L] = 8.1$, $\mathsf{V}[L] = 6$ when $\mu = 6.1$. Clearly, not serving jobs in the period they arrive has a non-negligible effect on the queue length. Clearly, being a bit flexible when dealing with customers can reduce the average queue length quite considerably.

WOULD VARIABILITY AFFECT the average queue length? We can suspect so, because when $a_i = 1$ and $c_i = 1$ for all i, and $L_0 = 0$, then $L_i = 0$ for all i.

However, if $c_1 = 5$ whenever i is a multiple of 5 and $c_i = 0$ elsewhere, then we do see queues appearing and disappearing. Let's use simulation to see how variability can influence the queue length process. The goal is now to make Fig. 2.5.2.

Before doing this, we need a way to quantify *relative* variability. To see the point, suppose the standard deviation of the time to serve a job is 1 minute. When the mean service time is 1 hour, we are inclined to call the service times very regular, while if the mean is just 1 minute, we would call it irregular. To capture this difference, we define the fundamentally important concept of *square coefficient of variation (SCV)* of a random variable X as

$$C_X^2 := \frac{\mathsf{V}[X]}{(\mathsf{E}[X])^2}.$$

Continuing with the simulation, we like to find a probability distribution for the services $\{c_i\}$ such that it has a specific mean μ and a scv c^2. A simple way to achieve this is to use a rv X such that $\mathsf{P}\{X = b\} = p$ and $\mathsf{P}\{X = 0\} = 1 - p$, and require that $\mathsf{E}[X] = pb = \mu$ and $C_X^2 = c^2$. Using that $\mathsf{V}[X] = \mathsf{E}\left[X^2\right] - (\mathsf{E}[X])^2$, we see that $C_X^2 = \mathsf{E}\left[X^2\right]/(\mathsf{E}[X])^2 - 1$. Rewriting this, we get $\mathsf{E}\left[X^2\right] = \mu^2(c^2+1)$, but we also know that $\mathsf{E}\left[X^2\right] = pb^2$. Solving for b and p in terms of μ and c^2 gives that $p = 1/(1+c^2)$ and $b = \mu/p = \mu(1+c^2)$.

With this construction, we can fix $\mathsf{E}[c_i] = \mu$, but vary the scv from one simulation to another. This next function implements the construction of a vector of rvs with the required properties.

```
92  def make_vector_mean_scv(mu, scv, num):
93      """Make a vector of length num with a given mean mu
94      and square coefficient of variation scv"""
95      p = 1 / (1 + scv)
96      b = mu / p
97      vec = b * rng.binomial(1, p, size=num)
98      return vec
```

Now we can run the simulations and make graphs for $c^2 = 0.5, 1, 2$ for the periodic service capacities. The arrivals are constant.

```
105 num = 5000
106 x = np.arange(0, num)
107 fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3), sharey=True)
108
109 ax1.set_title("Constant arrivals")
110 ax1.set_xlabel("time")
111 ax1.set_ylabel("Queue length")
112 a = make_vector_mean_scv(mu=6, scv=0, num=num)  # constant arrivals
113 for scv in (2, 1, 0.5):
114     c = make_vector_mean_scv(mu=6.5, scv=scv, num=num)
115     L = compute_queue_length(a, c, L0=0)
116     ax1.plot(x, L, linewidth=0.5, label=f"$c^2 = {scv}$")
117 ax1.legend()
```

To see the effect of variability in both the arrivals and services, we make a second plot with the next code, and save the result to file. Fig. 2.5.2 shows

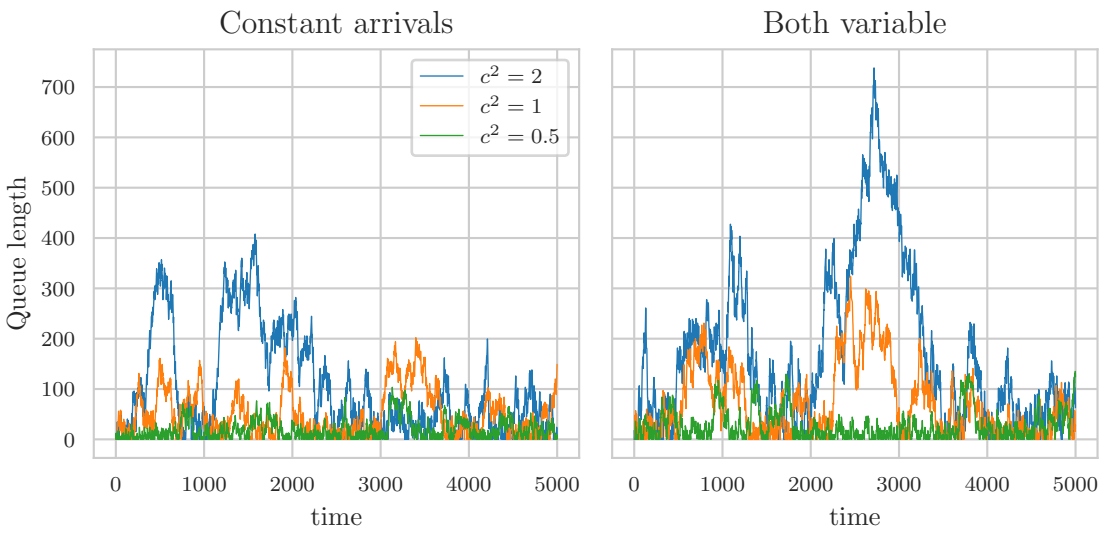


FIG. 2.5.2: *The influence of variability on queue length. In the left panel the arrivals are constant, but the number of services per period is variable with SCVs c^2 as indicated in the legend. In the right panel, we have variability in both arrivals and services. Thus, when there is variability in both arrivals and services, the queue length seems to become yet more variable. (Note that the scales on the y-axes are the same in the left and right panel.)*

clearly that variability has quite an influence on the queueing behavior: the larger the SCV, the larger the excursions of the queue length.

```
ax2.set_xlabel("time")
ax2.set_title("Both variable")
for scv in (2, 1, 0.5):
    a = make_vector_mean_scv(mu=6, scv=scv, num=num)
    c = make_vector_mean_scv(mu=6.5, scv=scv, num=num)
    L = compute_queue_length(a, c, L0=0)
    ax2.plot(x, L, linewidth=0.5, label=f"$c^2 = {scv}$")

fig.tight_layout()
fig.savefig('../figures/queue-discrete-time-scv.pdf')
```

In conclusion, when capacity is tight, or when relative variability in arrivals or services is large, queues are large. Hence, if we want to improve the performance of a queueing system, we basically have four options, increase the capacity, reduce the load (or demand), reduce the scv of the arrivals, or reduce the scv of the services. Of course, these options are not exclusive, for instance, we can try to reduce the variability of the service process and add some capacity at the same time.

IN OUR LAST example we consider some queues in sequence. For instance, to get your grade for this queueing course, several queueing stations are involved. First, I grade the exams (and handle all changes that might occur during the perusal). Then I mail a csv file with the grades to the secretary to have it uploaded to some database. The secretary also makes a print of this file which I have to sign because I am the examiner of the course. This paper is then sent to a general office that logs it to show that all internal processes have been in line with the legal obligations of the university. In

abstract terms, there are three stations (me, the secretary, and back office), and at least I am involved twice, once for the grading and a second time for the signing. More generally, many, if not most, service systems consist of a sequence of servers that have to be passed (multiple times) to complete a service from the perspective of a customer.

To get some insight into such systems, we can simulate a chain of 4 stations in tandem, which means that jobs arrive at station 1, are sent to station 2, and so on, until they leave the network after station 4. Jobs are assumed not to visit the same server, nor are lost or dropped somewhere half way the network.

Since in a tandem network the departures of station i are fed as arrivals into station $i+1$, we have to update the function to compute the queue lengths a bit.

```
135 def compute_queue_length(a, c, L0=0):
136     """Compute departures d and queue length L for
137     arrivals a,  capacities c and starting level L0"""
138     L = np.zeros_like(a)
139     d = np.zeros_like(a)
140     L[0] = L0
141     for i in range(1, len(a)):
142         d[i] = min(c[i], L[i - 1] + a[i])
143         L[i] = L[i - 1] + a[i] - d[i]
144     return d, L  # return d too
```

We next generate the arrivals at station 1, and the service capacities at each of the stations. Pay attention to the capacities: observe that station 2 has the slowest server. Note that if we use Poisson distributed service capacities, an integer amount of jobs is served each period, but the expected number served can be fractional.

```
151 labda, L0 = 6, 0
152 num = 5000
153 x = np.arange(num)
154
155 rng = np.random.default_rng(3)
156 a = rng.integers(labda - 1, labda + 2, size=num)
157 c1 = rng.poisson(labda + 1, num)
158 c2 = rng.poisson(labda + 0.1, num)
159 c3 = rng.poisson(labda + 1.5, num)
160 c4 = rng.poisson(labda + 1.0, num)
```

It remains to make the plots.

```
164 fig, ax = plt.subplots(figsize=(5, 3))
165 ax.set_xlabel("time")
166 ax.set_ylabel("Queue length")
167
168 d1, L1 = compute_queue_length(a, c1)
169 ax.plot(x, L1, linewidth=0.5, label="Q1")
170
171 d2, L2 = compute_queue_length(d1, c2)
172 ax.plot(x, L2, linewidth=0.5, label="Q2")
```

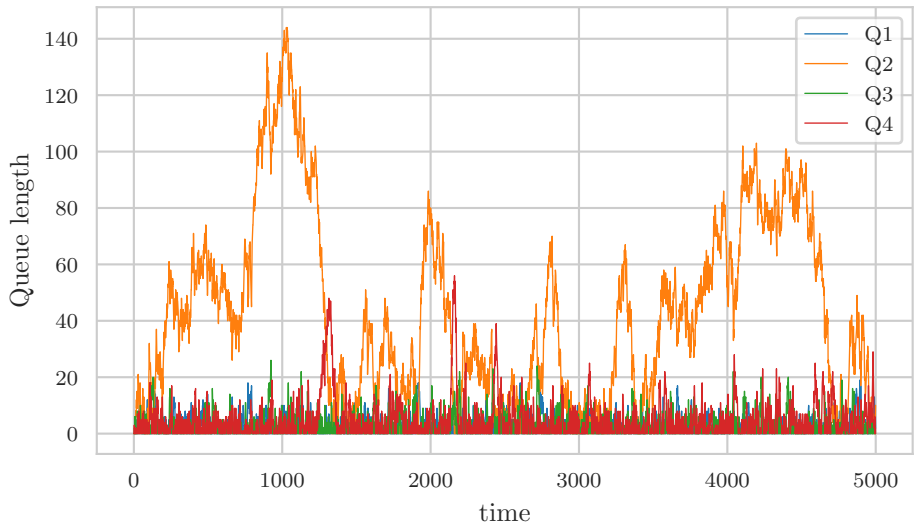


FIG. 2.5.3: *A queueing network of 4 stations in tandem.*

```
173
174  d3, L3 = compute_queue_length(d2, c3)
175  ax.plot(x, L3, linewidth=0.5, label="Q3")
176
177  d4, L4 = compute_queue_length(d3, c4)
178  ax.plot(x, L4, linewidth=0.5, label="Q4")
179
180  plt.legend()
181  fig.tight_layout()
182  fig.savefig('../figures/queue-discrete-network.pdf')
```

From Fig. 2.5.3 we learn that the station with the slowest server typically has the most jobs in queue in front of it. This brings us to an interesting idea. If we want to improve a queueing network, we should try of course to spend our (hard-earned) money on the bottleneck. Now, in many practical situations it is not directly evident what station is the bottleneck, because it can be difficult to obtain real good estimates of the number of services that a server can do per day. However, one clue to find the bottleneck station is by looking at the queue lengths in front of each station: the station with the largest queue is a good candidate for being a bottleneck.

There are many reasons why this is difficult. The capacity may depend on the person at the station on a certain day. There are disturbances due to raw materials missing or whether some machine at the station broke down or not. In short, in practical situations, there are many messy details that disturb the assembly of high quality data.

TF 2.5.1. A queueing system is stable when the arrival rate is not larger than the service rate.

TF 2.5.2. The scv of a random variable X is the same scv of αX if α a positive scalar.

TF 2.5.3. The parking meters at the universities are redesigned. A study has shown that the average amount of time that individuals need to spend to pay at the parking meter has gone down. Claim: as long as the interarrival distribution stays the same, the average queue length at the parking meter will be smaller.

TF 2.5.4. Claim: When $X \sim \text{Exp}(\lambda)$ its scv is larger than 1.

TF 2.5.5. Consider a queue with an exponential interarrival distribution and a service time distribution uniform on $[0, A]$. Claim:

$$C_s^2 = \frac{1}{3}.$$

Ex 2.5.6. A queueing system is under periodic review, i.e., at the end of each period the queue length is measured. Jobs arriving at period k cannot be served in period k and the system cannot contain more than K jobs; in other words, jobs are blocked when the system is too full. Develop code to simulate $\{L_k\}$ and compute the amount of jobs lost per period, and the fraction of jobs lost after simulating for T periods.

Ex 2.5.7. All items that are produced by a machine are tested. With probability p an item turns out to be faulty, and is returned to the queue for repair. The production time of new and faulty items is equal. Supposing items cannot be served in the period they arrive, develop a set of recursions to simulate $\{L_k\}$.

Ex 2.5.8. Sometimes items need rework. Suppose a fraction p of items does not meet the quality requirements after the first pass at a machine, but requires a second pass to repair the problems. Assume that repair jobs need half the service time of new jobs and are served with priority over new jobs. Develop the recursions.

Ex 2.5.9. A tandem queue with blocking is a production network with two production stations in tandem and blocking: the server at station 1 is not allowed to produce more than the amount M station 2 can maximally contain. We assume also that work jobs moves first from station 1 to station 2 before jobs from station 2 leave. Thus, besides the regular conditions, we need to impose $c_{k+1}^1 \leq M - L_k^2$. Find recursions.

Ex 2.5.10. Consider the discrete-time model specified by (2.1.1). We assume that a_k is a *batch* of jobs arriving in period k *after* the departures d_k have left. Provide a simulation to estimate $\mathsf{P}\{L \leq m\}$ for a situation in which the first job of the batch sees $L_{k-1} - d_k$ jobs in the system, the second sees $L_{k-1} - d_k + 1$ jobs, and so on.

3

CONSTRUCTION OF SIMPLE CONTINUOUS TIME STOCHASTIC PROCESSES

In Section 3.1 we focus on constructing queueing processes in continuous time. This contruction shows that a queueing system can be characterized by just a few elements; Section 3.2 develops highly useful notation for this purpose. Section 3.3 demonstrates how to implement the recursions in python and make plots. However, there are certain processes of interest that are not simply found in terms of recursion, such as the number of jobs $L(t)$ in the system as a function of time. Section 3.4 develops some new ides, such as heapqueues, and code to simulate $L(t)$ and some other somewhat more complicated queueing processes. Next, the models of these two sections are clean and nice, but do not allow to model and simulate really complicated systems such as stochastic systems under control, or multi-server queues whose servers have different server speeds. For this we extend in Section 3.5 the use of heap queues of Section 3.4 to discrete event simulation.

3.1 QUEUEING PROCESS IN CONTINUOUS TIME

In Section 2.1 we modeled time as progressing in discrete chunks. However, we can also model queueing systems in continuous time, so that jobs can arrive at any moment in time and have arbitrary service times. In this section, we develop a set of recursions to construct the waiting times of jobs served in the sequence in which the jobs arrive, i.e., according to the FIFO scheduling rule.

LET'S IMAGINE THAT a machine starts working on a one-hour job at 9 am, and there is no job in queue. When the next job arrives before 10 am, this second job has to wait in queue until the first job finishes at 10 am. If, however, this second job arrives after 10 am, it finds the server free, so that its service can start right at the moment of arrival.

More generally, suppose we are given an (ordered) sequence of *arrival times* $\{A_k\}$ and a set of iid *service times* $\{S_k\}$, such that job k arrives at time A_k and needs an amount S_k of service. Then we can compute the sequence of departure times $\{D_k\}$ from the recursions

Note that we assume that job k 'reveals' its service time at the moment it arrives.

$$\tilde{A}_k = \max\{A_k, D_{k-1}\}, \qquad D_k = \tilde{A}_k + S_k = \max\{A_k, D_{k-1}\} + S_k, \tag{3.1.1}$$

where $\tilde{A}_k$ is the time a job moves from the queue to the server. Why is this true? The service of job k cannot start before it arrives, hence, $\tilde{A}_k \geq A_k$. Moreover, job k cannot leave the queue before job $k-1$ left the server. Thus, the service of job k can also not start before D_{k-1}.

Once A_k and D_k are known, we can compute the *sojourn time* and the *waiting time* as

$$J_k = D_k - A_k, \qquad W_k = \tilde{A}_k - A_k = J_k - S_k;$$

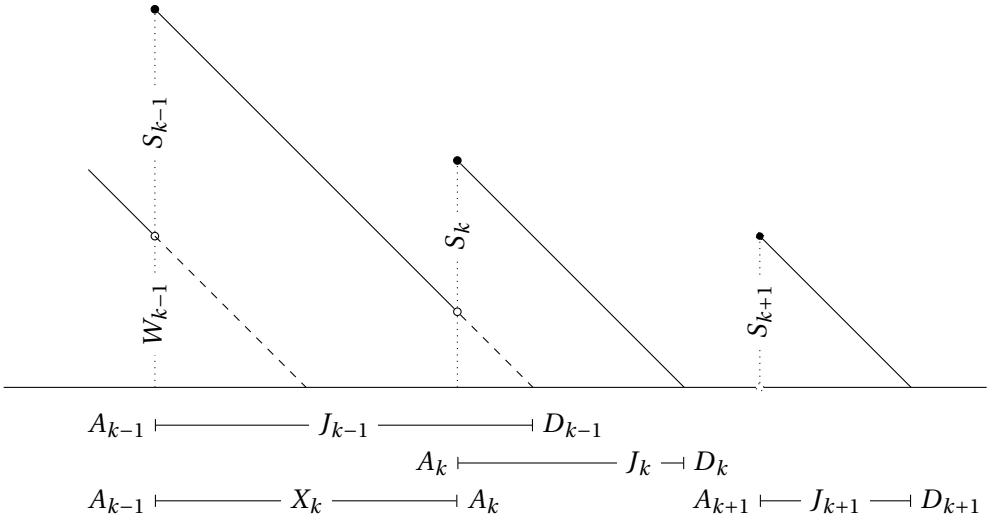


FIG. 3.1.1: *Construction of the single-server queue in continuous time. The sojourn time J_k is the time job k remains in the system, hence equals the sum of the waiting time W_k and the service time S_k. The virtual waiting time process V is shown by the solid lines with slope -1. Note that right after the arrival of job k, the virtual waiting time $V(A_k) = J_k$.*

thus, W_k is the time the job spends in queue (but not at the server), and J_k is the total amount of time job k spends in the system. Fig. 3.1.1 shows how the above concepts relate to each other.

WHILE IT IS often simple to measure arrival times in practical situations, it is a bit harder to generate $\{A_k\}$ directly with a simulator. Instead, in simulation we let a random number generator produce a set of iid rvs $\{X_k\}$ that represent the *inter-arrival times* between jobs. The arrival times can then be computed recursively with the rule

$$A_k = A_{k-1} + X_k, \qquad A_0 = 0. \tag{3.1.2}$$

Interestingly, if $\{X_k\}$ and $\{S_k\}$ are given, we don't need to first compute $\{A_k\}$ and $\{D_k\}$ via (3.1.1) to find the sojourn times and waiting times. In fact, suppose that job $k-1$ has to wait a time W_{k-1} in queue and then adds its service time S_{k-1} to the waiting time, so that its sojourn time $J_{k-1} = W_{k-1} + S_{k-1}$. If a time X_k elapses between the arrival time of job $k-1$ and k, then we can see from Fig. 3.1.1 that job k has to wait in queue,

$[x]^+ := \max\{x, 0\}$

$$W_k = [J_{k-1} - X_k]^+, \qquad J_k = W_k + S_k, \qquad W_0 = 0. \tag{3.1.3}$$

If we only need J_k or W_k then the next rules suffice:

$$J_k = [J_{k-1} - X_k]^+ + S_k, \qquad J_0 = 0, \tag{3.1.4}$$

$$W_k = [W_{k-1} + S_{k-1} - X_k]^+, \qquad W_0 = 0. \tag{3.1.5}$$

Henceforth, we make the implicit assumption that $\{X_k\}$ are iid, $\{S_k\}$ are iid, and $\{X_k\}$ are independent of $\{S_k\}$.

WE WILL SEE later that we need the state of the system at arbitrary moments in time, not just at arrival moments. The number of arrivals $A(t)$ during the interval $[0, t]$ can be computed from $\{A_k\}$ as

$$A(t) = \max\{k : A_k \le t\} = \sum_{k=1}^{\infty} \mathbb{1}_{A_k \le t}. \tag{3.1.6}$$

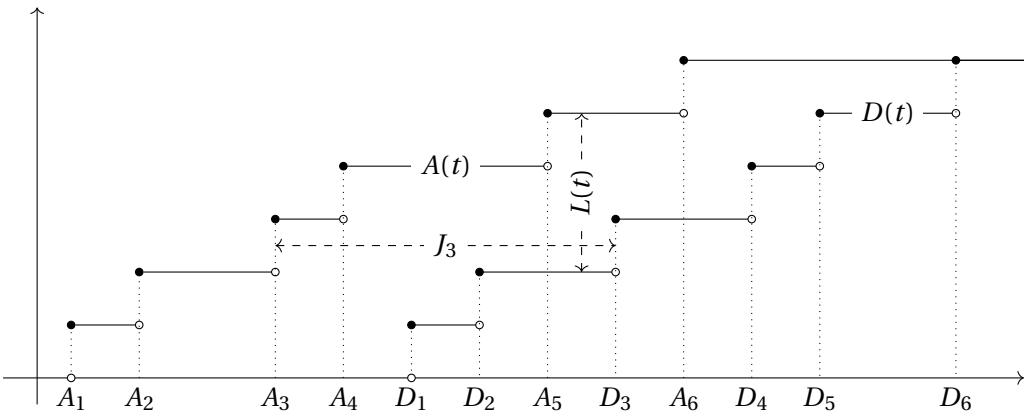


FIG. 3.1.2: *Relation between the arrival process* $\{A(t)\}$, *the departure process* $\{D(t)\}$, *the number in the system* $\{L(t)\}$ *and the sojourn times* $\{J_k\}$. *Observe that the arrival and departures times are intertwined.*

A function f is right-continuous if, for all x, $f(x) = f(x+) := \lim_{y \downarrow x} f(y)$.

Observe that the function $t \to A(t)$ is right-continuous.

In turn, we can retrieve the arrival times $\{A_k\}$ from the process $\{A(t)\}$. For instance, if we know that $A(s) = k-1$ and $A(t) = k$, then the arrival time A_k of the kth job must lie somewhere in $(s, t]$. Specifically,

$$A_k = \min\{t : A(t) \geq k\}, \quad A_0 = 0.$$

And if we have $\{A_k\}$, the inter-arrival times follow from $X_k = A_k - A_{k-1}$. So, with the above recursions and definitions, we can convert into each other the arrival times, the inter-arrival times and the number of arrivals as a function of time. It just depends on what our starting data is to obtain the rest.

Likewise, for the number of jobs $\tilde{A}(t)$ that departed from the queue up to time t and and the *departure process* $\{D(t)\}$ we have the relations

$$\begin{aligned} \tilde{A}(t) &= \max\{k : \tilde{A}_k \leq t\}, & \tilde{A}_k &= \min\{t : \tilde{A}(t) \geq k\}, \\ D(t) &= \max\{k : D_k \leq t\}, & D_k &= \min\{t : D(t) \geq k\}. \end{aligned}$$

Once we have the arrival and departure processes, it is easy to compute the *number of jobs in the system* at time t as, see Fig. 3.1.2,

$$L(t) = A(t) - D(t) + L(0), \tag{3.1.7}$$

where $L(0)$ is the number of jobs in the system at time $t = 0$; typically we assume that $L(0) = 0$. Recalling that in a queueing system, a job can either be in queue or in service, we distinguish between the number of jobs in the system $L(t)$, the number of jobs in queue $Q(t)$, and the number in service $L_s(t)$. Clearly, $L(t) = Q(t) + L_s(t)$ and $L_s(t) = \tilde{A}(t) - D(t) = L(t) - Q(t)$.

It is important to realize that the queue length process $\{Q(t)\}$ at general time moments t can be quite different from the queue length process $\{Q(A_k-)\}$ as observed by arriving jobs.

Observe that we write A_k-, and not A_k; we need to be careful about left and right limits at jump epochs.

Finally, the *virtual waiting time process* $\{V(t)\}$ is the time a job would have to wait if it would arrive at time t. In other words, the virtual waiting time is the waiting time observed by jobs arriving virtually (but not in reality). To construct $\{V(t)\}$, we simply draw lines that start at points (A_k, J_k)

[3.1.6]

and have slope -1, until the lines hit the x-axis. Once at $y = 0$, the virtual waiting time remains there until an arrival occurs, cf., Fig. 3.1.1. Here is the expression

See [3.1.6].

$$V(t) = [J_{A(t)} - (t - A_{A(t)})]^+; \tag{3.1.8}$$

the ingenious use of $A(t)$ can take some time to absorb!

FINALLY, WE NOTE that queueing and production-inventory systems are very similar, cf., Fig. 3.1.3. When a job arrives in the queueing system, the virtual workload $V(t)$ increases by the service time of the job. When a demand arrives in the inventory system, the inventory $I(t)$ decreases by the demand size of the customer. Like this, customer demands in the production-inventory system are job service times in the queueing system. Hence, in the figure, the demand size D_1 of the first customer corresponds to a production time of duration $S_1 = D_1$, and so on. Note that as long as the on-hand inventory level suffices to cover demand, customers do not have to wait to receive their product, but their demands spawn production times at the machine (a server) that replenishes the consumed items.

In a production-inventory system, a machine is switched on and off to replenish the inventory at a finite production rate. This is different from 'normal' inventory systems in which replenishments arrive as batches.

Assume now that the inventory process is controlled by an order-up-to policy: produce (refill the inventory) as long as the inventory level is below S and stop otherwise. Then the figure shows that the inventory level $I(t)$ is equal to $S - V(t)$, where $V(t)$ is the virtual waiting time of the related queueing system.

The figure shows in more general terms that in queueing systems or inventory systems, there is always 'something' or 'somebody' waiting. Items in the inventory of a supermarket are produced ahead and 'wait' until being consumed by customers. In a queueing system, customers are waiting while their product is being 'produced' by the server, and when there are no jobs, the server is idle and waits for jobs to arrive. Thus, queueing and inventory theory focus on waiting times, either by customers, servers, or items, hence both are related branches of (applied) probability theory and stochastic operations research.

TF 3.1.1. Claim: with our notation, the following mapping is correct:

$$A_k : \mathbb{N} \to \mathbb{R}, \quad \text{job id (integer) to arrival time (real number).}$$

TF 3.1.2. Claim: the number of arrivals $A(t)$ during $[0, t]$ can be defined as $\min\{k : A_k \geq t\}$.

TF 3.1.3 (3.1)**.** The waiting time of the third job is correctly represented in the figure below.

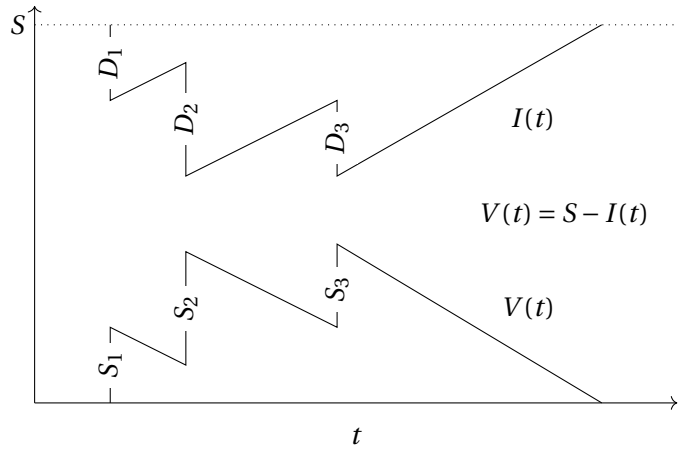
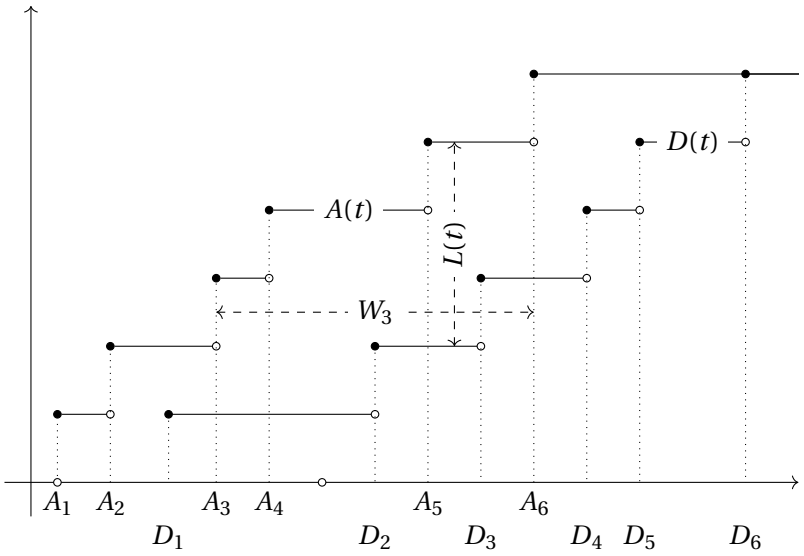


FIG. 3.1.3: *The relation between a production-inventory system and queueing. Here $I(t)$ models the evolution of the inventory level in an inventory system, while $V(t)$ shows the virtual workload, and S is the order-up-to level. When a customer requires D_1 items, say, it takes a server of a time $S_1 = D_1$ to produce these items. Thus, demands at the inventory system convert into production times in a queueing system. When the inventory is always replenished to level S, then the shortage of the inventory level relative to S, i.e., $S - I(t)$, becomes the workload $V(t)$ for the server in terms of amount of items or production time.*



TF 3.1.4. Claim: $A_{A(t)} = t$ for all t and $A(A_n) = n$ for all n.

Ex 3.1.5. Show that $L(A_k-) > 0 \iff A_k \le D_{k-1}$.

Ex 3.1.6. Explain (3.1.8) for the virtual waiting time at time t.

3.2 KENDALL'S NOTATION

As is apparent from Sections 2.1 and 3.1, the construction of a queueing process for a single station involves three main elements: the distribution of job inter-arrival times, the distribution of the service times, the number of servers present to process jobs, and the number (batch) of jobs that arrive and can be served at once.

To characterize the type of queueing process it is common to use *Kendall's abbreviation* $A^B/Y^C/c/K$, where A is the distribution of the iid inter-arrival times, Y the distribution of the iid service times, c the number of servers, and K the system size, i.e., the total number of customers that can be simultaneously present in the system, whether in queue or in service. It is implicit that $K \geq c$ for any queueing system. When K is not specified, the system capacity is unbounded, but when it appears in the acronym, queue length is bounded. When at an arrival epoch multiple jobs can arrive simultaneously (like a group of people entering a restaurant), we say that a *batch of jobs* arrives. Likewise, the server can work in batches, for instance, an oven can process multiple jobs at the same time. The letter B denotes the (distribution of the) batch size at arrival moments and C the (distribution of the) size of the service batch. When $B \equiv 1$ or $C \equiv 1$, i.e., single batch arrivals or single batch services, we suppress the B or C in Kendall's formula. Finally, in this notation, it is assumed that the service discipline is FIFO.

The meaning of K differs among authors. Sometimes it stands for the capacity of the queue, not the entire system. In this book K corresponds to the system's size.

For example, the size of groups of people entering a restaurant.

The most important inter-arrival and service distributions are the exponential distribution, denoted with the shorthand M, and the general distribution, denoted with G. We write D for a deterministic (constant) random variable.

M := Markov or Memoryless

With the implicit assumption that the first moment is finite

A FEW IMPORTANT examples are the following queueing processes: the $M/M/1$ queue in which inter-arrival times and service times are exponentially distributed; the $M/G/1$ queue with exponential inter-arrival times and general service times; and the $G/G/c$ queue in which the inter-arrival and service times are not specified.

A model that is often used to determine the number of beds needed in (a ward of) a hospital is the $M/M/c/c$ queue. The motivation is as follows. Practice tells us that patient inter-arrival times are memoryless, hence exponentially distributed. Data of patients treatment times shows that these times are often quite well described by an exponential distribution. Next, there are c beds available. Clearly, as each bed can serve just one patient, when all c beds are occupied, the hospital is 'full'. Thus, the maximal number of jobs in the system is the same as the number of servers available.

We can represent the discrete-time model that corresponds to the recursion (2.1.1) as a $D^B/D^C/1$ queue in which $a_k \sim B$ and $c_k \sim C$.

TF 3.2.1. Consider an $M^2/G/c/K$ queue, with $K \geq c$. Claim: exactly one of the following statements is false.

- Jobs arrive in pairs of two.
- Service times can be deterministic.
- The queue shared among c parallel servers cannot contain more than K jobs.

TF 3.2.2. In the $D/M/1$ jobs have deterministic service times.

TF 3.2.3. Consider a check-in desk at an airport. There is one desk that is dedicated to business customers. However, when it is idle (i.e., no business customer in service or in queue), this desk also serves economy class

customers. The other c desks are reserved for economy class customers. The queueing process as perceived by the economy class customers can be modeled as an $M/M/(c+1)$ queue.

3.3 DISTRIBUTION OF WAITING TIMES

We can determine the waiting time of the nth job in a single-server queue by means of the recursion $W_k = [W_{k-1} + S_{k-1} - X_k]^+$. That is, for given random deviates $\{X_k\}$ and $\{S_k\}$ we can compute the *number* W_k. If want to get information about the *distribution* of W_k, we need to do some more work. One way is to use simulation and repeat the recursions many different times with different seeds to make a histogram of the observed values of W_k. There is however, another way that is directly based on the tools for probabilistic arithmetic we developed in Section 1.4. Let us show some code how to we can do this; Fig. 3.3.1 will be the result of our effort.

TO START, WE load `pytictoc`, besides the regular libraries, to measure the running times of certain parts of the code. The module `random_variables.py` contains our code to handle the probabilistic arithmetic.. (We hide the lines that deal with `seaborn` on how to make the LaTeX plots.)

```
import numpy as np
import matplotlib.pyplot as plt
from pytictoc import TicToc

import random_variable as rv
```

To compute W_k we use the function $[x]^+ = \max\{x, 0\}$. We also instantiate a `tictoc` object.

```
def Plus(x):
    return max(x, 0)


tic = TicToc()
tic.tic()
```

As explained in Section 1.4 an `RV` object uses a dictionary to characterize a rv: the keys of the dict represent the support, the values the associated probabilities. Observe from the numbers below that the mean service time is larger than the mean inter-arrival time. This is on purpose.

```
X = rv.RV({3: 1 / 3, 4: 1 / 3, 5: 1 / 3})
S = rv.RV({4: 1 / 3, 5: 1 / 3, 7: 1 / 3})
```

The next few lines do the work to compute the distribution of W_{30}. We set the initial arrival such that $\mathsf{P}\{A_0 = 0\} = 1$, and then, by writing `A += X`, we compute the pmf of the rv $A + X$. Thus, the first call `A += X` computes the pmf of A_1. Since the system starts empty, the first job does not have to wait, which we implement by setting $\mathsf{P}\{W_1 = 0\} = 1$. In the for loop, we

update A_k according to the rule $A_k = A_{k-1} + X_k$ and the waiting time with $W_k = [W_{k-1} + S_{k-1} - X_k]^+$. We start the for loop at 2, because the lines before the loop compute the pmf of A_1 and W_1. The call to `tic.toc()` prints the time that elapsed between the 'tic' and the 'toc'. For later purposes, you should note the line that computes the departure times. We explain later why this way of computing the departure times is *wrong*.

```
horizon = 30
A = rv.RV({0: 1})  # Arrival time A_0
A += X  # Arrival time A_1
W = rv.RV({0: 1})  # Waiting time W_1
# Mind that we start at 2 instead of 1
for i in range(2, horizon):
    A += X
    W = rv.apply_function(Plus, W + S - X)

D = A + W + S

tic.toc()
```

Let us now estimate the distribution of W by means of simulation. We compute from $W_{30} = 0$ a number of `num_runs` times and store the outcomes in the array `nth_waiting_time`. In the computations we let the departure times tag along. We use small letters for the variables to avoid a name clash with the capitals like `X` we used earlier.

```
num_runs = 1000
nth_waiting_time = np.zeros(num_runs)
nth_departure = np.zeros(num_runs)
rng = np.random.default_rng(3)
for n in range(num_runs):
    x = X.rvs(horizon, rng)
    s = S.rvs(horizon, rng)
    x[0] = s[0] = 0

    a = x.cumsum()
    w = 0
    for i in range(1, horizon):
        w = Plus(w + s[i - 1] - x[i])
    nth_waiting_time[n] = w
    nth_departure[n] = a[i] + w + s[i]

# check KPIs
print(D.mean(), nth_departure.mean())
print(D.var(), nth_departure.var())

tic.toc()
```

Making the plots follows the same pattern as we did earlier. First we plot the cdfs of the departure times. The keyword `cumulative=True` converts a histogram to a cdf.

Recall, a histogram is an (approximation of a) pmf

```
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, figsize=(6, 3), sharey=True)
ax1.set_title("Departure times")
```

```
ax1.set_xlabel("Time")
ax1.hist(
    nth_departure,
    bins=30,
    density=True,
    cumulative=True,
    histtype='step',
    align="right",
    color='k',
    lw=0.5,
    label="sim",
)
ax1.plot(
    D.support(), [D.cdf(k) for k in D.support()], c='k', lw=0.75, label="exact"
)
ax1.legend()
```

And next the waiting times.

```
ax2.set_title("Waiting times")
ax2.set_xlabel("Time")
ax2.hist(
    nth_waiting_time,
    bins=30,
    density=True,
    cumulative=True,
    histtype='step',
    align="right",
    color='k',
    lw=0.5,
    label="sim",
)
ax2.plot(
    W.support(), [W.cdf(k) for k in W.support()], c='k', lw=0.75, label="exact"
)
```

The last step is to save the plot to file.

```
fig.tight_layout()
fig.savefig("../figures/waiting_time_distribution.pdf")
```

Finally, to plot the pmf instead of the cdf, we change the `cdf` in `pmf` and remove the lines with `cumulative=True`. Running the code again gives Fig. 3.3.2.

YOU MIGHT WONDER why we should have a preference for ploting the cdf instead of the pmf of the waiting and departure times; it seems that the pmf is easier to interpret. However, this is not true: the pmf can be an ill-behaved object. For instance, let us take the support of X as $\{3, 4.1, 5\}$ instead of $\{3, 4, 5\}$ as earlier, and leave the rest of the parameters untouched. Plotting the pmf in the same way as for Fig. 3.3.2, we now obtain Fig. 3.3.3.This looks quite a bit different from the densities in Fig. 3.3.2, to say the least.

If you doubt the correctness of the compuations, just run the same code but plot the cdfs instead; the pathology disappear.

Another interesting question is what would happen if we would compute the departure times with the following code, which is the implementation of the recursion $D_k = \max\{D_{k-1}, A_k\} + S_k$.

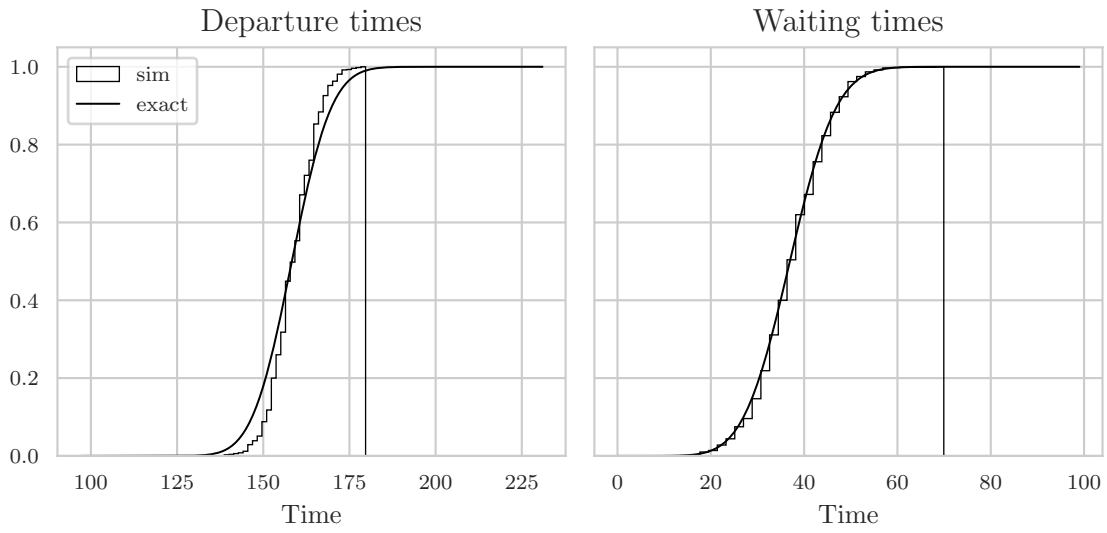


FIG. 3.3.1: *The cumulative distribution of the departure and waiting time of job* 30 *with the inter-arrival and service times as mentioned in the text. Observe that the distribution shifts to the right because the queue is not stable. The results of theory and simulation are similar, but there seems to a be a bit of a difference between the plot of the theoretical values for the cdf of departure times and the simulations. The main text explains why, and what the error is.*

```
A = rv.RV({0: 1})
D = rv.RV({0: 1})
for i in range(1, horizon):
    A += X
    D = rv.compose_function(max, D, A) + S
```

In fact, this is wrong for a subtle reason. The code in `random_variable.py` assumes that the rvs are *independent*, but D_k and A_{k-1} are dependent! To see this, observe from the left panel of Fig. 3.3.1 that $\mathsf{P}\{D_{30} \leq 150\} > 0$, but $\mathsf{P}\{D_{30} \leq 150 | A_{30} > 151\} = 0$ while still $\mathsf{P}\{A_{30} > 151\} > 0$. This also explains the conceptual mistake in our earlier code `D = A + W + S`, and why the graphs of the 'theoretical' cdf and the ecdf in the left panel of Fig. 3.3.1 do not agree. By inspecting this carefully, we see that the ecdf obtained by simulation starts to increase later than the cdf of D but increases faster. The variance of the simulated departure times are apparently smaller. In conclusion, as A_{30} and W_{30} are dependent, D_{30} cannot be computed as simply as the code suggests. In fact, the computation of the cdf of D_k requires a careful build up of the dependency structure of the rvs involved.

Dependent is the same as not independent.

The mistake lies not in the code of `RV` itself, but in how we use *the code.*

ecdf := empirical cdf

But why then does our compuation works in the code that implements the recursion $W_k = [W_{k-1} + S_{k-1} - X_k]^+$? This is because W_{k-1} *is* independent of S_{k-1} and X_k.

A nice python package that implements dependence in a correct way is `lea`.

Keep in mind that stochastic dependence is tricky and can be easily overlooked. To prevent making such mistakes (when it is important) it helps to check the results against simulation. If the results do not agree, there is an error.

TF 3.3.1. Claim: This code prints the mean departure time.

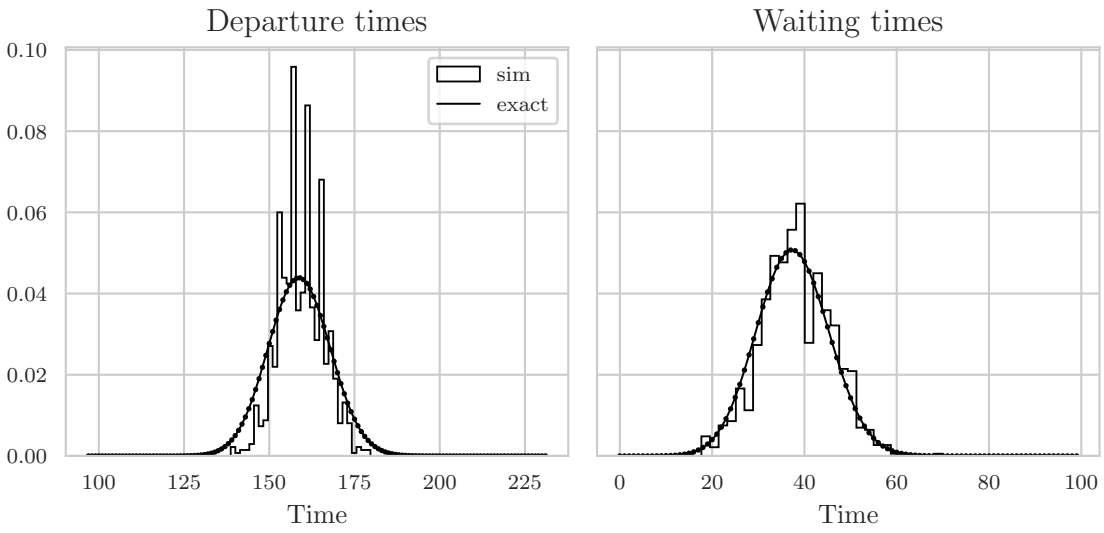


FIG. 3.3.2: *The pmfs of the departure and waiting time of job* 30 *with* X *concentrated on* $\{3,4,5\}$. *Compare this to Fig. 3.3.3.*

```python
import numpy as np

rng = np.random.default_rng(3)
labda, mu = 3, 4
num = 10
X = rng.exponential(scale=1 / labda, size=num)
Z = rng.exponential(scale=1 / mu, size=num)
X[0] = Z[0] = 0
A = X.cumsum()

D = np.zeros_like(X)
for i in range(1, num):
    D[i] = max(A[i], D[i - 1]) + Z[i]

print(D.mean())
```

TF 3.3.2. Consider a queue in continuous time with inter-arrival times $\{X_k\}$, service times $\{S_k\}$, waiting times $\{W_k\}$ and sojourn times $\{J_k\}$. Claim: we can compute the waiting times and sojourn times with the following recursion:

$$W_k = [J_{k-1} - X_k]^+, \qquad J_k = W_k + S_k.$$

TF 3.3.3. Claim: This code uses the recursion $D_k = \max\{D_{k-1}, A_k\} + S_k$ in the correct way to compute the probability distribution of $\{D_k\}$.

```python
A = rv.RV({0: 1})
D = rv.RV({0: 1})
for i in range(1, horizon):
    A += X
    D = rv.compose_function(max, D, A) + S
```

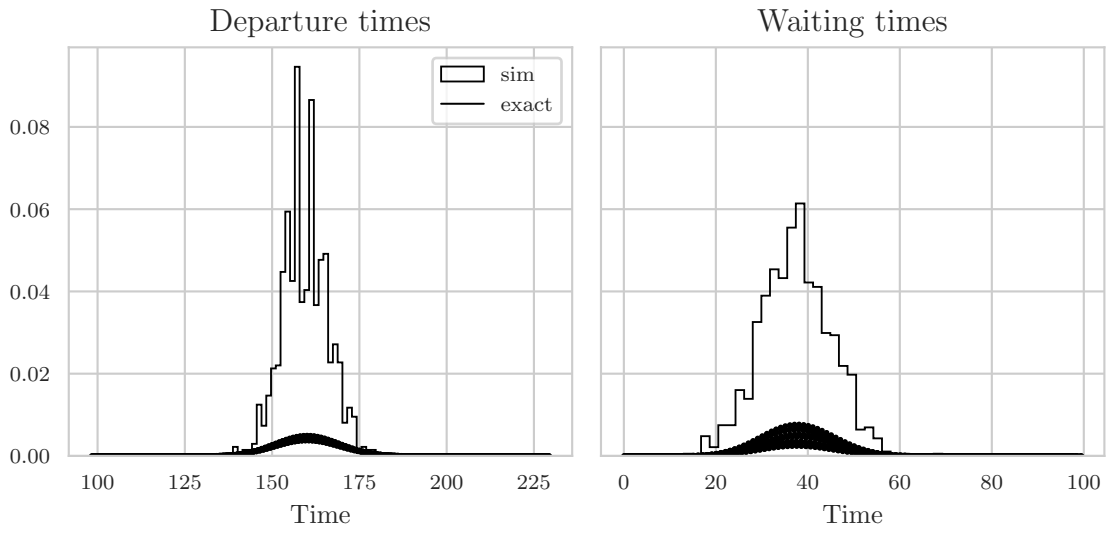


FIG. 3.3.3: *The pmfs of the departure and waiting time of job* 30 *with X with support* $\{3, 4.1, 5\}$ *instead of* $\{3, 4, 5\}$*; the rest of the parameters are as in Fig. 3.3.1. Note how pathological the pmf of the theoretical distribution behaves.*

3.4 SIMULATIONS IN CONTINUOUS TIME

With the recursions of Section 3.1 we can simulate simple queueing systems in continuous time. In this section we implement these recursions in python and show how to build a graph of the system length process L and the virtual waiting time process V. This is not entirely straightward, though: to get around the problems, we need an *event stack*, which is actually a queueing process operating under a specific scheduling rule. More generally, event stacks form the heart of every discrete event simulator you will ever use. In Section 3.5 we will deal with the details of discrete event simultion; here we start simple.

Thus, we use a queue, i.e., the event stack, to simulate queueing processes.

THE DYNAMICS OF the single server queue in continuous time is given by the recursions in (3.1.1). The relating code is simple, but there is a small detail we should solve. If we were to compute the total offered amount of work, then the natural thing is to call `S.sum()`. However, this includes `S[0]`, while actually job 0 is never served. So, we set `S[0]=0`, which is in line with the choice to set `X[0]=0`. This induces a slight error though: `S.mean()` includes still $S[0]$, so we divide by one job too much; to repair we should compute `S.mean() * len(S) / (len(S) -1)`, but we choose to neglect this small error.

Be aware, in critical software you should not use such tricks.

The next code computes first the departure times D, and then the sojourn times J and waiting times W. Note how easy this is. Calling statistics is trivial too.

```
import numpy as np

rng = np.random.default_rng(3)
labda, mu = 3, 4
num = 10
X = rng.exponential(scale=1 / labda, size=num)
S = rng.exponential(scale=1 / mu, size=num)
```

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

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

J = D - A  # sojourn times
W = J - S  # waiting times

# Statistics
print(J.mean(), J.var())
```

WITH THE RECURSIONS (3.1.1) it is apparently easy to compute the sojourn and waiting times, but it is less simple to compute the number of jobs in queue or in the system at arrival moments, for instance. The problem is that the arrival and departure times are all intertwined, see Fig. 3.1.2. Before discussing how to get around this, we first show how to compute the time-average of the number of jobs in the system, as this is still relatively easy and develops some formulas that we will need for the proof of Little's law, cf., Section 4.4.

Fig. 3.1.2 clarifies that the number of jobs $L(s)$ in the system at time s is equal to all jobs $A(s)$ that arrived minus the jobs $D(s)$ that departed during $[0,s]$, assuming that $L(0)=0$. Therefore, if we run a simulation for n jobs in total, the time average number of jobs during the simulation must be $D_n^{-1}\int_0^{D_n} tL(s)\,\mathrm{d}s$, because the last job departs at time D_n. Another way to look at the integrand $L(s)$ is to count all jobs whose arrival times lie before s and whose departure time lie after s, that is, $L(s)=\sum_{k=1}^{n}\mathbb{1}_{A_k\le s\le D_k}$. Therefore, the time-average number L must satisy,

$$\begin{aligned}\bar{L} &= \frac{1}{D_n}\int_0^{D_n} L(s)\,\mathrm{d}s = \frac{1}{D_n}\int_0^{D_n}\sum_{k=1}^{n}\mathbb{1}_{A_k\le s\le D_k})\,\mathrm{d}s = \frac{1}{D_n}\sum_{k=1}^{n}\int_0^{D_n}\mathbb{1}_{A_k\le s\le D_k})\,\mathrm{d}s\\ &\overset{1}{=} \frac{1}{D_n}\sum_{k=1}^{n} J_k = \frac{n}{D_n}\frac{1}{n}\sum_{k=1}^{n} J_k = \frac{n}{D_n}\bar{J},\end{aligned}$$

where 1 follows from the fact that the integral $\int_0^t \mathbb{1}_{A_k\le s\le D_k}\,\mathrm{d}s = J_k$ if $D_k\le t$. In code, to get an estimate for the average system length, just sum all sojourn times and divide by the time the last job leaves the system.

```
print(J.sum() / D[-1])
```

LET US NEXT turn towards designing an algorithm to compute the number L_k of jobs in the system as seen by job k upon its arrival. Clearly, as time A_k is the arrival time of job k, the number of jobs that arrived during $[0,A_k]$ is precisely k. Thus, if we subtract all jobs that departed before A_k, we have the number of jobs in the system, i..e, $L_k = k-\sum_{i=1}^{k}\mathbb{1}_{D_i\le A_k}$. Now this formula works, but has a terrible algorithmic performance. To see this, note that for job k this formula compares A_k with the departure times of all earlier jobs. For k jobs in total, we therefore need $1+2+\cdots k = k(k+1)/2$ comparisons, which has $O(k^2)$ complexity. When k is a million, the square is a really large number indeed.

The next code is much more efficient. We keep a *pivot*, i.e., a pointer, to the last departure before job k. Suppose this pivot is i, in other words, job i was the last job that departed before A_k, then, for L_{k+1} we start searching for departures from job $i+1$ onward.

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

A MORE FLEXIBLE way to compute $\{L_k\}$ is by means of an *event stack*, but to understand what an event stack is, we first need to discuss *heap queues*.

A heap queue stores elements in a list such that they can be retrieved in accordance to some ordering rule. The following piece of code demonstrates the idea. We *push* (add) a tuple containing an age and an animal name to the *heap*, and we *pop* from the heap to get the animals in order of age. Here is an example; study the output carefully.

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

>>> heap = []

>>> heappush(heap, (25, "Turtle"))
>>> heappush(heap, (21, "Horse"))
>>> heappush(heap, (20, "Lion"))
>>> heappush(heap, (18, "Cat"))
>>> heappush(heap, (14, "Dog"))

>>> print(heappop(heap)) # Why is it the dog?
(14, 'Dog')
>>> print(heappop(heap)) # Why is it the cat?
(18, 'Cat')
>>> heappush(heap, (23, "Elephant")) # Add Elephant

>>> while heap:
...     e = heappop(heap)
...     print(e)
...
(20, 'Lion')
(21, 'Horse')
(23, 'Elephant')
(25, 'Turtle')
```

Note that anytime we pop from the heap, the heap becomes one element shorter, and with pushing the heap becomes longer. Moreover, we can push animals, here an elephant, at our convenience, without having to wait until the heap is empty. As such, a heap queue is the ideal (and efficient) data structure to support simulation because we can push and pop events in any sequence we like, but we have the guarantee that events are popped in the correct sequence of time.

WITH HEAP QUEUES WE have an appropriate data structure to plot the virtual waiting time and the system length processes in Fig. 3.4.1. In passing we encounter some other useful and generic ideas.

Suppose we set $L = 0$ at time 0, and add 1 to L for each arrival and subtract 1 for each departure, then we have the system length if the arrival times and departures times are sorted in time. For this to work, we form events of type (t, Δ) where t is an arrival or departure time and $\Delta \in \{-1, 1\}$ to correspond to a departure or arrival, respectively. To sort the events, we form two lists of events and merge them.

There is a technical, but important, detail involved. The function `heapq.merge` returns a generator, not a list, but `heapq.heapify` expects a list instead of a generator. To convert the generator obtained from `heapq.merge` we apply `list` to this generator, and then pass the resulting list to `heapq.heapify`.

```
arrivals = [[t, 1] for t in A[1:]]
departures = [[t, -1] for t in D[1:]]
heap = list(heapq.merge((arrivals, departures)))
heapq.heapify(heap)
```

With this heap we can already plot the system length, but for the virtual waiting time we need some additional code. Recall that $V(t) = [J_{A(t)} - (t - A_{A(t)})]^+$, and that $A(t) = \sum_{k=1}^{\infty} \mathbb{1}_{A_k \leq t}$. The following code would work to find the last arrival before time t:

```
k = 1
while A[k] <= t:
    k += 1
```

However, the efficiency of this code is terrible. Instead, we should stick the rule that says 'Thou shalt use binary search to find an element in a sorted sequence'. We therefore use `np.searchsorted` to efficiently find the index k in an array A such that $A_{k-1} \leq t < A_k$ for some given value t. In [3.4.5] we explain why the next code does the job.

```
def At(t, side='right'):
    """
    The right or left continuous version of A(t), depending on side,
    A(t) = max {k : A_k \geq t}
    A(t-) = A(t) - 1.
    """
    return max(np.searchsorted(A, t, side=side) - 1, 0)
```

Note that we include the option `side` to provide us with the left-continuous version $A(t-)$; $A(t)$ is defined as right-continuous.

The virtual waiting time follows now directly from its recursive definition.

```
def V(t, side='right'):
    """
    The right or left continuous virtual waiting time at t, depending on SIDE
```

```
    """
    k = At(t, side)
    return max(J[k] - (t - A[k]), 0)
```

All is in place to make Fig. 3.4.1. We have two panels `ax1` and `ax2` on top of each other, and we set the labels at the axes.

```
fig, (ax1, ax2) = plt.subplots(nrows=2, figsize=(6, 3), sharex=True)
ax1.set_ylabel("Virtual time~$V$")
ax2.set_ylabel("Length~$L$")
ax2.set_xlabel("time")
ax2.set_yticks([0, 1, 2, 3, 4, 5])
```

For the sojourn and waiting times we draw triangles.

```
for i in range(1, num):
    ax1.fill([A[i], A[i], D[i]], [0, J[i], 0], c='green', alpha=0.2)
    ax1.fill([A[i], A[i], A[i] + W[i]], [0, W[i], 0], c='blue', alpha=0.2)
```

Setting the opacity `alpha` to 0.2 gives nice results for overlapping triangles.

Next is the code to plot L and V. Pay attention to the fact that we use $V(t-)$ (in code `V(now, 'left')`). Lines 5 and 7 plot vertical dotted lines for visual clarity. Formally speaking, these lines do not belong to the virtual waiting time or system length processes.

```
L, past = 0, 0
while heap:
    now, delta = heapq.heappop(heap)
    ax1.plot([past, now], [V(past), V(now, 'left')], c='k', lw=0.75)
    ax1.plot([now, now], [V(now, 'left'), V(now)], ":", c='k', lw=0.75)
    ax2.plot([past, now], [L, L], c='k', lw=0.75)
    ax2.plot([now, now], [L, L + delta], ":", c='k', lw=0.75)
    L += delta
    past = now
```

To compile the figure with LaTeX and write to file we follow the same steps as we used to make Fig. 2.5.1.

TF 3.4.1. We have a single-server queue where the interarrival and service times have a general distribution. Claim, when $L(T) = 0$ at time T, then

$$\int_0^T L(s)\,\mathrm{d}s = \int_0^T \sum_{k=1}^{A(T)} 1\{A_k \le s < D_k\}\,\mathrm{d}s$$
$$= \sum_{k=1}^{A(T)} \int_0^T 1\{A_k \le s < D_k\}\,\mathrm{d}s = \sum_{k=1}^{A(T)} J_k.$$

TF 3.4.2. For a queue where the interarrival and service times have a general distribution the virtual waiting time process $\{V(t), t \ge 0\}$ satisfies

$$V(t) = [J(A_{A(t)}) + (A_{A(t)} - t)]^+.$$

TF 3.4.3. Claim: the following code pops the element: `(1.9, 'Long')`.

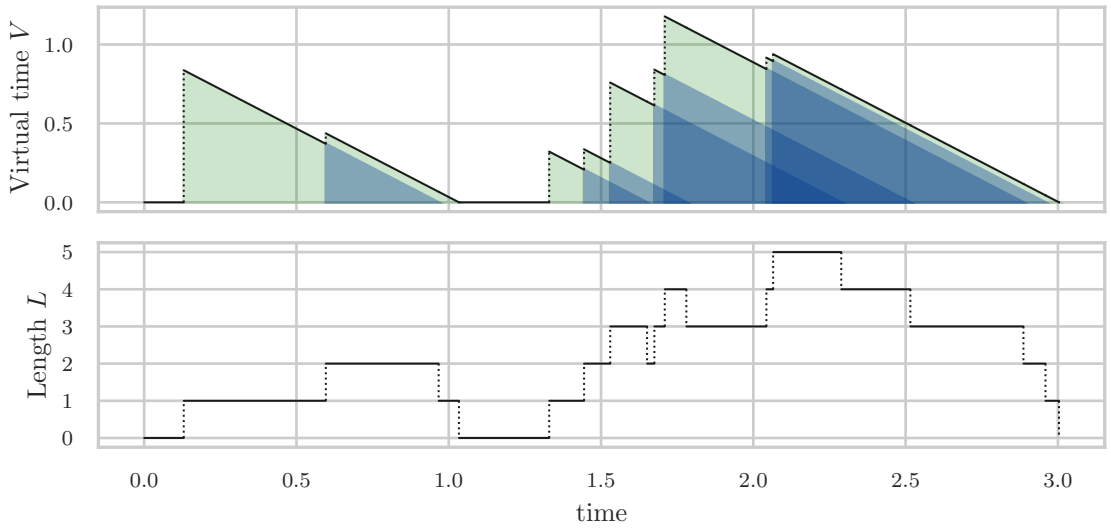


FIG. 3.4.1: *The top panel shows a graph of the sojourn times and the virtual waiting time, the lower panel the number of jobs in the system. The darker the blue, the more jobs the system contains at that moment.*

```
from heapq import heappop, heappush
heap = []
heappush(heap, (1.9, "Long"))
heappush(heap, (1.7, "Egg"))
heappush(heap, (1.3, "Zeno"))
print(heappop(heap))
```

Ex 3.4.4. Suppose that we want to compute the time-average of L for a time $t < D_n$, i.e., a time that lies before the departure time of the last job of the simulation. How would you compute this?

Ex 3.4.5. Why is $\max\{k : A_{k-1} \leq t < A_k\} = 1 + A(t)$?

3.5 DISCRETE EVENT SIMULATIONS

The queueing and inventory systems in the previous sections were relatively simple to simulate. If, however, we need to analyze more difficult situations we need discrete event simulators, and to demonstrate how that works we will build such a simulator for a single station multi-server queue. We first provide a recursion in mathematical terms, and then discuss its shortcomings. This serves as motivation to set up a simulator with classes and an event stack.

LET US NEXT construct a multiserver FIFO queue in which the service of the first job in line starts when a server becomes free; if a server is free when a job arrives, the job's service starts right away.

Suppose there are c servers available, each with its own waiting line, like in a supermarket. When job k arrives, it sees a waiting time $w_k(i)$ at line i; write $w_k = (w_k(1), \ldots, w_k(c))$ for the vector of waiting times. In other words, if job k would join line i, it would have a waiting time of $w_k(i)$. Of course,

the job selects the line with the shortest waiting time. Thus, it selects line $s_k = \arg\min\{w_k(i) : i = 1,\ldots,c\}$.

This is not necessarily the same as the shortest queue.

To formulate a recursion for w_k, let $e(i)$ be the ith unit vector, and $\mathbf{1} = (1,\ldots,1)$. The waiting time of job $k-1$ becomes $W_{k-1} = w_{k-1}(s_{k-1})$, and, in analogy with (3.1.3), the vector w_k updates as

A 1 at place i and zeros elsewhere.

$$s_{k-1} = \underset{i\in\{1,\ldots,c\}}{\arg\min}\{w_{k-1}(i)\}, \qquad w_k = [w_{k-1} + S_{k-1}e(s_{k-1}) - X_k\mathbf{1}]^+,$$

where $[\cdot]^+$ applies element-wise.

It is useful to analyze the algorithmic complexity of this algorithm. For job $k-1$, we need to find the minimum in w_{k-1}, and compute and subtract $X_k\mathbf{1}$. The number of computational operations for this is $2c$, as w_k and $\mathbf{1}$ contain c elements. For a simulation with N jobs, the total amount of operations is therefore $2cN$. However, by using a different implementation, the complexity can be reduced to $N\log_2 c$.

With event stacks.

WE NOW TURN to developing a simulator for the multi-server queue step by step. We organize the code by means of classes. There are many reasons for this, but one is that this allows us to represent 'things' of the 'real world' in code that mimics the properties and behavior of the real world object.

The Server is our first class. We implement this as a dataclass because the server is a very simple class with just two attributes: an id and a service rate so that we can allow for speed differences between servers. We add the class method `__lt__` to be able to compare two servers based on their relative rates, and select a free server based on its rate. Specifically, since we prefer a fast server over a slow server, we use the inequality as shown.

A lower service rate means that jobs are served at a slower rate.

server.py
```
from dataclasses import dataclass


@dataclass
class Server:
    id: int
    rate: float

    def __lt__(self, other):
        return self.rate > other.rate
```

The next class is the Servers class and represents a pool of free servers. We subclass Servers from a list so that we can use this list for the heapq algoritms to push and pop servers. When a job leaves the queue, Servers will assign a server to the job based on the preferences expressed by the `__lt__` method of the Server class. We push to the servers heap any server that becomes idle after finishing a job,. Finally, we add the convenience methods `num_free` and `is_server_available` as these are easier to understand and read than their implementations in terms of `len`.

servers.py
```
from heapq import heappop, heappush


class Servers(list):
```

```
    def push(self, server):
        heappush(self, server)

    def pop(self):
        return heappop(self)

    def num_free(self):
        return len(self)

    def is_server_available(self):
        return self.num_free() > 0
```

The Job class speaks mostly for itself. The service time will be the job's load divided by the speed of the server that serves the job. As the service time depends on the server, and since the servers can have different speeds, we cannot compute the service time at the start of the simulation. The job keeps a reference to the server that handles the job. The other attributes are used for the statistics at the end of the simulation. We implement the sojourn time and waiting time as a `property`, as this allows us to address them in the same way as, for instance, the departure time. The `__lt__` method determines which job to select from the queue when a service can start. As it is now, we select jobs in FIFO sequence. If, however, we would prefer a LIFO queue, we only have to reverse the direction of the inequality. Another simple rule is Shortest Processing Time First. To use this, change the inequality to this `self.load < other.load`.

LIFO := last in first out

There is one detail. We might want to store jobs in a set or a dict, and this requires to be able to distinguish one job from another. For this reason we give the option `eq=True` to the data class and include a `__hash__` method.

job.py

```
from dataclasses import dataclass

from server import Server


@dataclass
class Job:
    id: int
    arrival_time: float
    load: float
    service_time: float = 0
    departure_time: float = 0
    queue_length: int = 0
    free_servers: int = 0
    server: Server = None

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

    @property
    def waiting_time(self):
        return self.sojourn_time - self.service_time

    def __repr__(self):
        return (
```

```
            f"{self.id},  {self.load}, {self.service_time},"
            f"{self.arrival_time}, {self.departure_time}"
        )

    def __hash__(self):
        return self.id

    def __lt__(self, other):
        return self.arrival_time < other.arrival_time
```

The fourth class is the Queue class. This is also a subclass from a list so that we can use the queue as a heap in itself, and use the `__lt__` method of the Job class to select a job from the queue and send that job to the servers to be served.

queues.py

```
from heapq import heappop, heappush


class Queue(list):
    def pop(self):
        return heappop(self)

    def push(self, job):
        heappush(self, job)

    def length(self):
        return len(self)

    def is_empty(self):
        return self.length() == 0
```

An Event is just a time and job attached to it. The `__lt__` method specifies how to order events. Clearly, this must be by time. We subclass the event class to an ArrivalEvent and a DepartureEvent, because during the simulation we need to distinguish the type of event.

event.py

```
from dataclasses import dataclass

from job import Job


@dataclass
class Event:
    time: float
    job: Job

    def __lt__(self, other):
        return self.time < other.time


class ArrivalEvent(Event):
    pass


class DepartureEvent(Event):
    pass
```

The events class is a heapqueue and sorts the events in time. In a way, once you realize a heapqueue offers the functionality to order, remove and insert events, the core of discrete event simulation is easy indeed.

events.py

```
from heapq import heappop, heappush


class Events(list):
    def push(self, event):
        heappush(self, event)

    def pop(self):
        return heappop(self)

    def is_empty(self):
        return len(self) == 0
```

The Statistics class does what its name says. We subclass it from a list, because with a list we can keep the sequence of the jobs in which the jobs depart from the server.

stats.py

```
class Statistics(list):
    def push(self, job):
        self.append(job)

    def num_jobs(self):
        return len(self)

    def mean_waiting_time(self):
        tot = sum(job.waiting_time for job in self)
        return tot / self.num_jobs()

    def mean_sojourn_time(self):
        tot = sum(job.sojourn_time for job in self)
        return tot / self.num_jobs()

    def mean_queue_length(self):
        tot = sum(job.queue_length for job in self)
        return tot / self.num_jobs()

    def mean_servers_free(self):
        tot = sum(job.free_servers for job in self)
        return tot / self.num_jobs()
```

Study it carefully.

The last class we need is the Simulator itself. We need references to the events, queue, stats and servers. The attribute `now` keeps track of the time: it points to the time of the event the simulator is currently treating. When starting the simulation, we send a list of jobs to the simulator which then pushes an ArrivalEvent to the event stack for each job.

When a job service starts, the `serve_job` method asks for a free server, and assigns this server to the job. Then it computes the job service time and departure time, and pushes a DepartureEvent to the event stack to inform the simulator later that a job is finished.

The `run` method starts by checking whether there are still events to do, and if so, it pops an event from the event stack. Then it retrieves the time

and stores it as `now`, and gets the job attached to the event. If the event is an ArrivalEvent, the job must have just arrived and can be served if a server is free, otherwise the job is queued. When the event is a DepartureEvent, the server that served the job becomes free and is pushed to the stack of free servers. The departing job is pushed to the statistics tracker. If there is still work in the queue, a service can start for the job that is at the head of the queue.

simulator.py

```
from event import ArrivalEvent, DepartureEvent


class Simulation:
    def __init__(self, events, queue, statistics, servers):
        self.events = events
        self.queue = queue
        self.stats = statistics
        self.servers = servers
        self.now = 0

    def initialize_jobs(self, jobs):
        for job in jobs:
            self.events.push(ArrivalEvent(job.arrival_time, job))

    def serve_job(self, job):
        server = self.servers.pop()
        job.server = server
        job.service_time = job.load / server.rate
        job.departure_time = self.now + job.service_time
        job.queue_length = self.queue.length()
        job.free_servers = self.servers.num_free()
        self.events.push(DepartureEvent(job.departure_time, job))

    def run(self):
        while not self.events.is_empty():
            event = self.events.pop()
            self.now, job = event.time, event.job
            if isinstance(event, ArrivalEvent):
                if self.servers.is_server_available():
                    self.serve_job(job)
                else:
                    self.queue.push(job)
            elif isinstance(event, DepartureEvent):
                self.servers.push(job.server)
                self.stats.push(job)
                if not self.queue.is_empty():
                    job = self.queue.pop()
                    self.serve_job(job)
            else:
                raise ValueError("Unknown event")
```

IT REMAINS TO actually try out the simulation. Later, in Section 5.1, we provide a set of formulas for the so-called $M/M/c$ queue which is a queueing process in which the iid inter-arrival times $X_i \sim \text{Exp}(\lambda)$, the iid service times $S_i \sim \text{Exp}(\mu)$, with $\mu > \lambda$, and there are c identical servers. These formulas give exact results for the expected waiting time and related KPIs, so this model serves as a perfect benchmark for our simulator. We use the next

code for the test. The code for the $M/M/c$ queue is contained in a separate python file with the name `mmc.py`; you can find it on github.

For the generation of the data for the simulation we follow the same logic as in Section 3.4.

test-mmc.py

```
"""This compares the result of the simulation of a multi-server queue
 to the exact results of an M/M/c queue.
"""

import numpy as np

import mmc
from events import Events
from job import Job
from queues import Queue
from server import Server
from servers import Servers
from simulator import Simulation
from stats import Statistics


def test_mmc():
    num = 10000
    labda, mu = 3, 4

    rng = np.random.default_rng(3)
    X = rng.exponential(scale=1 / labda, size=num)
    X[0] = 0
    A = X.cumsum()
    loads = rng.exponential(scale=1 / mu, size=num)
    jobs = [Job(id=i, arrival_time=A[i], load=loads[i]) for i in range(1, num)]

    servers = Servers()
    for i, rate in enumerate([1, 1]):
        servers.push(Server(id=i, rate=rate))

    sim = Simulation(Events(), Queue(), Statistics(), servers)
    sim.initialize_jobs(jobs)
    sim.run()

    mm2 = mmc.MMC(labda, mu, c=2)
    print("W: ", sim.stats.mean_waiting_time(), mm2.EW)
    print("J: ", sim.stats.mean_sojourn_time(), mm2.EJ)
    print("Q: ", sim.stats.mean_queue_length(), mm2.EQ)


test_mmc()
```

Here is the output. The simulator seems to pass the test.

```
W:  0.044123468095817915 0.04090909090909091
J:  0.2952268030097956 0.2909090909090909
Q:  0.1467146714671467 0.12272727272727273
```

Let us now demonstrate with a few examples how flexible this simulation environment actually is.

SUPPOSE WE HAVE three different servers with rates 1, 0.1 and 0.01, respectively, and we like to select the fastest server whenever multiple servers are free. As it turns out, our simulation supports this already, not a single line has to change. The Servers class acts as a heap, and popping occurs with the preference as expressed by the `__lt__` method of the Server class.

Here are some interesting experiments.

multiple-speeds.py

```
"""Simulate a multi-server queue with different speeds and compare
to an M/M/C queue."""
from collections import Counter

import numpy as np

import mmc
from events import Events
from job import Job
from queues import Queue
from server import Server
from servers import Servers
from simulator import Simulation
from stats import Statistics


def multiple_speeds():
    num = 10000
    labda, mu = 3, 3.1

    rng = np.random.default_rng(3)
    X = rng.exponential(scale=1 / labda, size=num)
    X[0] = 0
    A = X.cumsum()
    loads = rng.exponential(scale=1 / mu, size=num)
    jobs = [Job(id=i, arrival_time=A[i], load=loads[i]) for i in range(1, num)]

    servers = Servers()
    for i, rate in enumerate([1, 0.1, 0.01]):
        servers.push(Server(id=i, rate=rate))

    sim = Simulation(Events(), Queue(), Statistics(), servers)
    sim.initialize_jobs(jobs)
    sim.run()

    mm1 = mmc.MMC(labda, mu, c=1)
    print("W: ", sim.stats.mean_waiting_time(), mm1.EW)
    print("J: ", sim.stats.mean_sojourn_time(), mm1.EJ)
    print("Q: ", sim.stats.mean_queue_length(), mm1.EQ)

    count = Counter([j.free_servers for j in sim.stats])
    print(count)


multiple_speeds()
```

This is the output when $\lambda = 3$ and $\mu = 4$.

The value in the code may be different from this, because I used the same code for different experiments.

```
W:  0.4178019854514783 0.75
J:  1.2386226229512027 1.0
Q:  1.2715271527152716 2.25
```

This is interesting, for several reasons. In the multi-server case the second and third server add just a little bit of service capacity. For this reason we compare the result to the $M/M/1$ queue, with just one server working at rate 1. We see that the average queue length is smaller in the multi-server station, but the sojourn time is larger. The reason must be that some jobs end up at a slow server. To see what fraction that is, we use the `Counter`. After simulating, we get this.

```
Counter({0: 8772, 1: 1136, 2: 91})
```

Obviously, a significant number of jobs gets assigned to a slow server.

We might wonder whether we should use the extra capacity offered by the slow servers. Perhaps we should only do this when the queueing times become really large. So, let's increase the load of the jobs by changing $\mu = 4$ to $\mu = 3.1$. We get the following results.

```
W:  2.0385311552319467 9.677419354838683
J:  2.971938896180861 9.999999999999973
Q:  6.165616561656166 29.03225806451605
Counter({0: 9615, 1: 357, 2: 27})
```

And now it becomes very clear that the extra capacity helps. Interestingly, the fraction of jobs served by the fastest server is *larger* than when the load is smaller.

Another interesting rule to select servers could be based on the cost of running the server. For instance, it may be that some servers use less energy than others, so that an operator has a preference for the cheaper servers. We will the analysis of this rule (and variations) to you to pursue.

IN SOME QUEUEING systems, some jobs have priority over others, like in a hospital. With the next job class we can implement this behavior. As a priority job is just a job with a priority, it suffices to subclass the job class from above and change the ordering rule. The ordering rule is simple: when jobs have the same priority, they should be sorted according to FIFO, otherwise according to priority (here lower priority is better).

priority-job.py

```
from dataclasses import dataclass

from job import Job
from server import Server


@dataclass(eq=False)
class PriorityJob(Job):
    priority: int = 0

    def __lt__(self, other):
        if self.priority == other.priority:
            return self.arrival_time < other.arrival_time
        else:
            return self.priority < other.priority
```

The queue class is already a heap, so needs no change.

Here is how to use it.

priority-jobs.py

```python
"""The effect of priority service in multi-server queues."""
from collections import defaultdict

import numpy as np

import mmc
from events import Events
from priority_job import PriorityJob
from queues import Queue
from server import Server
from servers import Servers
from simulator import Simulation
from stats import Statistics


def priority_jobs():
    num = 100000
    labda, mu = 3, 3.2

    rng = np.random.default_rng(3)
    X = rng.exponential(scale=1 / labda, size=num)
    X[0] = 0
    A = X.cumsum()
    loads = rng.exponential(scale=1 / mu, size=num)
    priorities = rng.binomial(1, 0.1, size=num)
    jobs = []
    for i in range(1, num):
        job = PriorityJob(id=i, arrival_time=A[i], load=loads[i])
        job.priority = priorities[i]
        jobs.append(job)

    servers = Servers()
    for i, rate in enumerate([1]):
        servers.push(Server(id=i, rate=rate))

    sim = Simulation(Events(), Queue(), Statistics(), servers)
    sim.initialize_jobs(jobs)
    sim.run()

    mm1 = mmc.MMC(labda, mu, c=1)
    print("W: ", sim.stats.mean_waiting_time(), mm1.EW)
    print("J: ", sim.stats.mean_sojourn_time(), mm1.EJ)
    print("Q: ", sim.stats.mean_queue_length(), mm1.EQ)

    wait = defaultdict(float)
    nums = defaultdict(float)
    for j in sim.stats:
        nums[j.priority] += 1
        wait[j.priority] += j.waiting_time
    for k in sorted(nums.keys()):
        print(f"Priority: {k}, EW: {wait[k] / nums[k]:.2f}")


priority_jobs()
```

And this is the result.

```
W:  4.6978119398356775 4.6875
J:  5.01121142029712 5.0
```

```
Q:  14.089830898308984 14.0625
Priority: 0, EW: 1.92
Priority: 1, EW: 29.38
```

For the population as a whole there is no difference in average waiting time (which is according to our intuition), but the lower priority jobs perceive much longer waiting times.

WITH THIS CODE environment you have all the tools available to simulate and analyze many different queueing (and inventory) systems.

For instance, when jobs have due-dates, we might want to serve the job that has the smallest due-date to minimize the probability of that job being late. Then we have to add a due-date attribute to jobs, and use that in the method `__lt__` to compare jobs. Of course, there are many other different scheduling rules possible.

A more complicated example is how to handle business and economy class customers at a check-in desk at an airport. Ofter there is one server strictly allocated to business class customers, and there are c, say, servers strictly used by the economy class customers. Another way to organize the server allocation could be like this. When the business server is free, this server takes just one economy customer in service (assuming there is such a customer in queue). When, after this service, there is still no business customer in queue, the business server serves another economy customer. Once a business customer arrives, the business server finishes the economy customer (if there is any in service), and then serves the business customer. We use a similar policy for serving business customers if there are free economy servers. The problem is, of course, to analyze the performance of this adapted policy. By how much do the waiting times of the business customers increase, and by how much do the waiting times of the economy class customers decrease? Perhaps we can remove a server, thereby making the tickets cheaper.

It may not be simple to make the necessary adaptions. Different customer queues may be necessary, we might have multiple stations, but the basic interaction with an event stack remains the same.

TF 3.5.1. Claim: this class implements a sorting rule in which jobs with the same priority are served in LIFO order.

```
class PriorityJob(Job):
    priority: int = 0

    def __lt__(self, other):
        if self.priority == other.priority:
            return self.arrival_time < other.arrival_time
        else:
            return self.priority > other.priority
```

TF 3.5.2. Claim: this part of simulator class handles the arrival of jobs correctly.

```
class Simulation:
    def run(self):
```

```
        while not self.events.is_empty():
            event = self.events.pop()
            self.now, job = event.time, event.job
            if isinstance(event, ArrivalEvent):
                self.queue.push(job)
                self.serve_job(job)
```

TF 3.5.3. Claim: this part of simulator class handles the serving of jobs correctly.

```
class Simulation:
    def serve_job(self, job):
        server = self.servers.pop()
        job.server = server
        job.service_time = job.load / server.rate
        job.departure_time = self.now + job.service_time
        job.queue_length = self.queue.length()
        job.free_servers = self.servers.num_free()
        self.events.push(DepartureEvent(job.departure_time, job))
```

TF 3.5.4. Claim: this part of simulator class handles the process of departing jobs correctly.

```
class Simulation:
    def run(self):
        while not self.events.is_empty():
            if isinstance(event, DepartureEvent):
                self.servers.push(job.server)
                if not self.queue.is_empty():
                    job = self.queue.pop()
                    self.serve_job(job)
```

Ex 3.5.5. Consult the website of RealPython to obtain a better understanding of

1. classes
2. dataclasses
3. heapq
4. deque

Ex 3.5.6. There are some advantages to subclassing Queue from `deque`. What are these?

4

FUNDAMENTAL TOOLS

With the tools developed in Chapter 2 and Chapter 3 we can *simulate* very general stochastic processes. While this is very useful, reasoning about (properties of) queueing and inventory systems is simpler with formulas and models; in this chapter we make a start with that.

The mathematical characterization of the *transient behavior* of stochastic systems is extremely complicated, so henceforth we focus on the *long-run time average* behavior. For this the arrival and service rate play a crucial role. In particular, a queueing or inventory system is only stable when demand can be served at a higher rate than they arrive. Once we have established the stability, we define performance measures such as the long-run average waiting time.

We next derive some results that are fundamental to analyze any stochastic process. These results are based on *sample-paths*, i.e., realizations of the simulation of such systems, and form an elegant and unifying principle between the constructions of Chapter 2 and Chapter 3 and the theoretical results we will consider henceforth. To illustrate the relations between all concepts we provide two mind-maps at the end of the chapter. Here we keep the discussion in these notes mostly at an intuitive level; we refer to El-Taha and Stidham Jr. [1998] for proofs and further background.

4.1 RATE, STABILITY AND LOAD

In this section, we develop a number of measures to characterize the performance of queueing systems in steady-state. In particular, we define the load, which is, arguably, the most important performance measure of a queueing system to check.

We first develop the concepts for a queueing process with batch and service sizes of 1, i.e. $B = C = 1$ in Kendall's formula. At the end we present the general formula.

WE FIRST FORMALIZE the arrival rate and departure rate in terms of the arrival and departure processes $\{A(t)\}$ and $\{D(t)\}$, cf., Section 3.1. The *arrival rate* is the long-run average number of jobs that arrive per unit time along a sample path, i.e.,

$$\lambda = \lim_{t\to\infty} \frac{A(t)}{t}.$$

This limit does not necessarily exist if $A(t)$ is some pathological function.

Likewise, the *departure rate* is

$$\delta = \lim_{t\to\infty} \frac{D(t)}{t}. \tag{4.1.1}$$

Henceforth, we assume that both limits exist, are finite, and $0 < \delta \leq \lambda < \infty$, where the middle inequality is true when the system starts empty, i.e., $L(0) = 0$, because in that case $D(t) \leq A(t)$.

FOR THE ARRIVAL RATE, suppose we are given a sequence of iid inter-arrival times $\{X_k\}$, each of which is distributed as the common rv X. Then, by the definition of A_n, $\sum_{k=1}^{n} X_k = A_n$. If $\mathsf{E}[X] < \infty$, then by the strong law, $A_n/n \to \mathsf{E}[X]$ as $n \to \infty$. The next theorem relates this to the limit of $A(t)/t$ as $t \to \infty$; this relation is not entirely evident because the first limit runs of over the natural numbers, while the second runs over the real numbers.

Theorem 4.1.1. Assume that $\lambda \in (0,\infty)$. Then, the existence of one of the next two limits implies the existence of the other, in which case,

$$\frac{A_n}{n} \to \lambda^{-1} > 0 \iff \frac{A(t)}{t} \to \lambda > 0.$$

Proof. Observe first that when $A(t) = n$ then $A_{A(t)} = A_n$, and therefore, $A_{A(t)}/A(t) = A_n/n$. By assumption at least one of the limits is positive, so $A(t)$ and A_n must increase without bound. *[3.1.4]*

Second, $A_{A(t)}$ is the arrival time of the last job *before* time t, hence $A_{A(t)} \le t$. This, in turn, implies that $A_{A(t)+1}$ is the arrival time of the first job *after* time t. Therefore, $A_{A(t)} \le t < A_{A(t)+1}$, hence for $A(t) > 0$,

$$\frac{A_{A(t)}}{A(t)} \le \frac{t}{A(t)} < \frac{A_{A(t)+1}}{A(t)} = \frac{A_{A(t)+1}}{A(t)+1}\frac{A(t)+1}{A(t)}.$$

If $A_n/n \to \lambda^{-1}$, then $t/A(t) \to \lambda^{-1}$, because $A(t) \to \infty \implies A(t)+1)/A(t) \to 1$.

If $A(t)/t \to \lambda$, then on the epochs A_k we must also have that $A(A_k)/A_k \to \lambda$. Noting that $A(A_k) = k$, it follows that $k/A_k \to \lambda$. □

Recall: $A(t)/t \to \lambda$ *as* $t \to \infty \iff \forall \epsilon > 0 : \exists s : \forall t > s : |A(t)/t - \lambda| < \epsilon$.

From this theorem it follows for the average inter-arrival time $\mathsf{E}[X]$ between two consecutive jobs that

$$\mathsf{E}[X] = \lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} X_k = \lim_{n\to\infty} \frac{A_n}{n} = \lim_{t\to\infty} \frac{t}{A(t)} = \frac{1}{\lambda},$$

i.e., the inverse of the arrival rate.

For the service rate of a *single* server, take S_k as the required service time of the kth job served by the server, so that $U_n = \sum_{k=1}^{n} S_k$ is the total service time of the first n jobs. Similar to the definition of $A(t)$, we let $U(t) = \max\{n : U_n \le t\}$ and define the *service*, or *processing*, rate of a single server as

$$\mu := \lim_{t\to\infty} \frac{U(t)}{t}, \quad 0 < \mu < \infty,$$

assuming that this limit exists. This provides us with the relation

$$\mathsf{E}[S] = \lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} S_k = \lim_{n\to\infty} \frac{U_n}{n} = \lim_{t\to\infty} \frac{t}{U(t)} = \frac{1}{\mu}.$$

Thus, the service rate μ of a single server is the *inverse* of $\mathsf{E}[S]$.

WITH THE ARRIVAL and service rate, we define for the single server queue the

$$\text{Server load } = \rho := \lambda\,\mathsf{E}[S] = \frac{\lambda}{\mu} = \frac{\mathsf{E}[S]}{EX},$$

and the

$$\text{Server utilization } = \delta \mathsf{E}[S] = \frac{\delta}{\mu}.$$

We need to distinguish between these two concepts when $\lambda > \mu$ or when arriving customers are blocked. In the first situation, the queue in front of the server increases beyond bound so that $\lambda > \mu = \delta$. In the second situation, the blocked jobs do not enter the system, hence are not served, hence $\lambda > \delta$. Bear in mind, however, that the load can exceed 1 (when $\lambda > \mu$), while the utilization is always ≤ 1.

Of course, the departure rate can never exceed the service rate.

In general we say that a system is *rate-stable* if

$$\lambda = \delta.$$

In words: the system is rate-stable whenever in the long run jobs leave the system just as fast as they enter the system. As in this case the load and the utilization are equal, we will use these terms interchangeably for rate-stable systems. Clearly, for rate-stability, it is necessary that $\lambda \leq \mu$.

It is easy to generalize the definition for the load. When each job contains $\mathsf{E}[B]$ items each of which requires $\mathsf{E}[S]$ time to serve, the work per job is $\mathsf{E}[S]\,\mathsf{E}[B]$. Similarly, when a server serves, on average, a batch of $\mathsf{E}[C]$ of items per service time, its service rate is $\mathsf{E}[C]/\mathsf{E}[S]$. Finally, when the station contains identical c servers, the station's capacity is c times are large as that of a single server. Thus, in this more general case,

$$\rho := \frac{\lambda \mathsf{E}[S]}{c} \frac{\mathsf{E}[B]}{\mathsf{E}[C]}.$$

The utilization has to change accordingly.

Finaly, we assume that service is *work-conserving*, which means that the server is not idle when there is at least one job in the system..

Thus, when service is work-conserving, capacity is not wasted when there are jobs in the system

WE CAN USE the memoryless property of the inter-arrival times of the $M/G/1$ queue to show that the expected busy time of the $M/G/1$ satisfies $\mathsf{E}[U] = \mathsf{E}[S]/(1-\rho)$. During the service time of the first customer that starts a busy time, an expected number $\lambda \mathsf{E}[S]$ new jobs arrive. As each of these jobs restarts the busy-period, the expected busy time started by the first jobs must be equal to the expected service time of this job plus all busy periods that are generated by the jobs that arrive during this service time, hence, $\mathsf{E}[U] = \mathsf{E}[S] + \lambda \mathsf{E}[S]\,\mathsf{E}[U]$. But from this, $\mathsf{E}[U] = \mathsf{E}[S]/(1-\rho)$, because $\rho = \lambda \mathsf{E}[S]$. Realize further that because inter-arrivals times are memoryless for the $M/G/1$ queue, the expected idle time $\mathsf{E}[I] = 1/\lambda$.

TF 4.1.1. Claim: If for a queueing system $\lim_{t\to\infty} A(t) = \infty$, then the system is not rate stable.

TF 4.1.2. Claim: $\lim_{t\to\infty} \frac{A(t)}{t} = \lim_{n\to\infty} \frac{A_n}{n}$.

TF 4.1.3. Claim: $\lambda = \delta \implies \rho = 1$, hence the queue is not stable.

Th 4.1.4. Can you make an arrival process such that $A(t)/t$ does not have a limit?

4.2 (LIMITS OF) EMPIRICAL PERFORMANCE MEASURES

In Section 3.1 we use the arrival process $\{A(t)\}$ and the service times $\{S_k\}$ to construct the waiting times $\{W_k\}$, sojourn times $\{J_k\}$, and the number in the system $\{L(t)\}$. If the queueing system is rate-stable, we can sensibly define several more long-run average performance measures. Recall that these definitions are the most used quantitative measures to express customer satisfaction.

DEFINE THE EXPECTED *waiting time in queue* as

$$\mathsf{E}[W] = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} W_k.$$

Note that this is the limit of waiting times as *observed by arriving jobs*: the first job has to wait W_1 in queue, the second W_2, and so on. The *distribution* of the waiting time as seen by arrivals can be found by counting:

We colloquially say that a statistic based on the sampling of arriving jobs is 'as seen by arrivals'.

$$\mathsf{P}\{W \le x\} = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \mathbb{1}_{W_k \le x}. \tag{4.2.1}$$

For the sojourn time J we use similar definition. The *average number of jobs* in the system as seen by arrivals is given by

$$\mathsf{E}[L] = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} L(A_k-), \tag{4.2.2}$$

since $L(A_k-)$ is the number of jobs in the system just before the arrival epoch of the kth job. Like (4.2.1), for the distribution of L,

$$\mathsf{P}\{L \le m\} = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \mathbb{1}_{L_k \le m}. \tag{4.2.3}$$

A RELATED SET of performance measures follows by tracking the system's behavior over time and taking the *time-average*.

Now we sometimes say that such performance measures are as 'seen by the server'.
Even though the symbols are the same, this expectation is not necessarily the same as (4.2.2).

Assuming the limit exists, we use (3.1.7) to define the *time-average number of jobs* as

$$\mathsf{E}[L] = \lim_{t\to\infty} \frac{1}{t} \int_0^t L(s)\,\mathrm{d}s. \tag{4.2.4}$$

The *time-average fraction of time the system contains at most m jobs* is defined as

$$\mathsf{P}\{L \le m\} = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathbb{1}_{L(s)\le m}\,\mathrm{d}s.$$

PROVING THE EXISTENCE of the above limits requires a considerable amount of mathematics. Here we sidestep all such fundamental issues, and simply assume that all is well-defined. The limiting random variables are known as the *steady-state* or *stationary* limits.

See Asmussen [2003] if you are interested.

IN THE NEXT chapters we will derive simple formulas for the above performance measures for a variety of queueing systems in steady-state. These formulas, and the relations between them, provide *structural insight* into the behavior of the systems, rather than just numbers. Hence, for many practical purposes, which is (should be?) based on insight, a steady-state analysis suffices.

TF 4.2.1. Claim: If the limit exists, then

$$\frac{1}{t}\sum_{k=1}^{A(t)} \mathbb{1}_{W_k \le x} \to \mathsf{P}\{W \le w\}, \quad t \to \infty.$$

TF 4.2.2. Consider a $G/G/1$ queue that is rate-stable. Claim: The distribution of the waiting times at arrival times can be sensibly defined as

$$\mathsf{P}\{W \le x\} = \lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} \mathbb{1}_{W_k \le x}.$$

TF 4.2.3. The departure process $\{D(t)\}$ can be computed from the set $\{D_k\}$ of departure times according to:

$$D(t) = \sum_{k=1}^{\infty} \mathbb{1}_{D_k \le t}.$$

Ex 4.2.4. Design a queueing system to show that the average number of jobs in the system as seen by the server can be very different from what customers see upon arrival.

Ex 4.2.5. If $L(t)/t \to 0$ as $t \to \infty$, can it still be true that $\mathsf{E}[L] > 0$?

4.3 RENEWAL REWARD THEOREM

In a one-dollar store, each item costs \$1. If every minute a customer arrives, and each customer buys exactly 10 items, this shop earns money at a rate of \$10 per minute. At another shop, customers spend variable amounts but on average \$10 per customer, and customers come in irregularly, but the average inter-arrival time between two customers is 1 minute. This other shop also earns money at rate of \$10 per minute. That both shops have the same earning rate is what the *renewal reward* theorem says. In this section we prove this extremely useful theorem and show how to apply it to queueing and inventory systems.

WE ARE GIVEN a sequence of strictly increasing epochs $0 = T_0 < T_1 < T_2 < \cdots$ such that $T_k \to \infty$ as $k \to \infty$, and a non-decreasing right-continuous (deterministic) function $t \to Y(t)$ with $Y(0) = 0$. Let $N(t) = \sum_{k=1}^{\infty} \mathbb{1}_{T_k \le t}$ count the number of epochs that occurred during $(0, t]$, and $X_k = Y(T_k) - Y(T_{k-1})$ be the increment of $Y(t)$ during $(T_{k-1}, T_k]$. Fig. 4.3.1 shows graphically how all concepts relate.

Here X_k is not necessary an inter-arrival time between two jobs.

Theorem 4.3.1 (Renewal Reward Theorem, $Y = \lambda X$)**.** Assume that the limit $\lambda = \lim_{t\to\infty} N(t)/t$ exists with $0 < \lambda < \infty$. Then, the limit $Y = \lim_{t\to\infty} Y(t)/t$ exists iff $X = \lim_{n\to\infty} n^{-1}\sum_{k=1}^{n} X_k$ exists, in which case, $Y = \lambda X$.

Proof. To see why this is true, we copy an idea of Section 4.1. Just as with the relation between $A(t)$ and $A_{A(t)}$, we have here that $T_{N(t)} \le t < T_{N(t)+1}$. As $Y(\cdot)$ is non-decreasing,

$$Y(T_{N(t)}) \le Y(t) \le Y(T_{N(t)+1}),$$

hence

$$\frac{N(t)}{t}\frac{Y(T_{N(t)})}{N(t)} \le \frac{Y(t)}{t} \le \frac{Y(T_{N(t)+1})}{N(t)+1}\frac{N(t)+1}{t}.$$

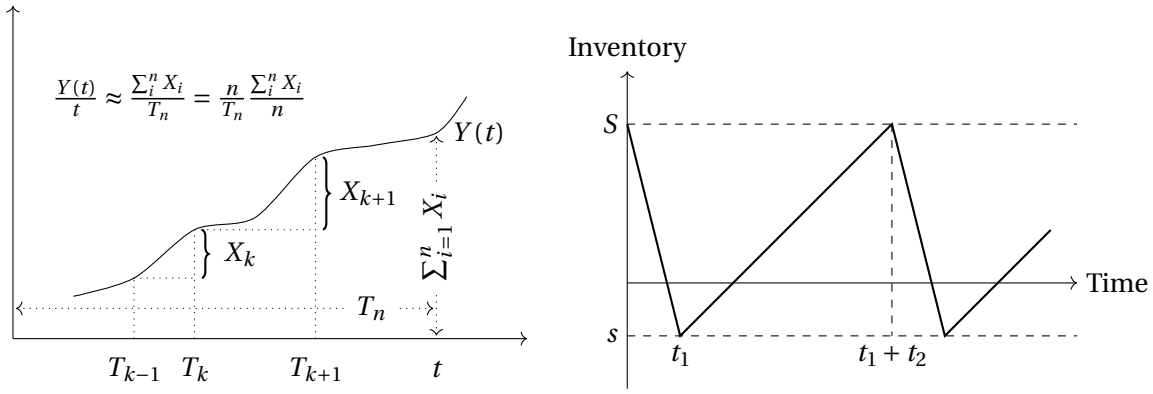


FIG. 4.3.1: *The left panel provides a graphical 'proof' of the renewal reward equation $Y = \lambda X$. The right panel shows a sample path of an inventory system with constant production capacity and a fixed demand rate. As the inventory is operated under an (s, S)-policy, it is a system that moves from cycle to cycle.*

Noting that $Y(T_{N(t)}) = \sum_{k=1}^{N(t)} X_k$, and using the assumption that $N(t)/t \to \lambda$ we conclude that $Y(t)/t \to Y$ iff $(N(t))^{-1}\sum_{k=1}^{N(t)} X_k \to X$. If one of these limits exists, the other exists, hence $Y = \lambda X$. □

THE RENEWAL REWARD theorem finds an easy application in the analysis of the *Economic Production Quantity* (EPQ) model which is an inventory system in which demand arrives at a fixed rate d and items are produced at constant rate r by a machine that can switch on and off. The inventory is under an (s, S) policy, the leadtime $L = 0$, the holding and backlogging cost are h and b per unit per time, and it costs K to switch on the machine. Fig. 4.3.1 shows a sample path of the inventory level. Let us use the ideas of renewal reward theory to find the policy parameters that minimize the long-run time average cost.

The EPQ model is a production inventory system with a constant demand rate.

Clearly, the inventory level repeats over time, so it suffices to study just one cycle that starts and stops when the inventory level hits S. From Fig. 4.3.1 it is simple to see that

$$t_1 = \frac{S-s}{d}, \qquad t_2 = \frac{S-s}{r-d}, \qquad T = t_1 + t_2 = \frac{S-s}{d}\frac{r}{r-d},$$

are the time to empty the inventory from level S to s (after which production switches on) and the time needed to replenish the inventory from level s to S (after which production switches off).

Let us next assume that $s \le 0 < S$. Then, writing T_+ (T_-) for the amount of time that the inventory is positive (negative) during a cycle, it follows that

$$\frac{T_+}{T} = \frac{S}{S-s}, \qquad \frac{T_-}{T} = \frac{-s}{S-s}.$$

Using that the holding and backlogging cost are proportional to the areas of the related triangles, the average cost of one cycle becomes

$$g = \frac{K}{T} + b\frac{-s}{2}\frac{T_-}{T} + h\frac{S}{2}\frac{T_+}{T}.$$

When $0 \le s < S$, $g = K/T + h(s+S)/2$, and when $s < S \le 0$, $g = K/T + b|s+S|/2$.

Substituting the expressions for T_-, T_+, and T in terms of s and S, taking the partial derivatives with respect to s and S and solving for the optimal values by setting $\partial g/\partial s = \partial g/\partial S = 0$ gives

$$g^* = \sqrt{2Kd}\sqrt{\frac{hb}{h+b}}\sqrt{1-\frac{d}{r}}, \qquad s^* = -\frac{g^*}{b}, \qquad S^* = \frac{g^*}{h}. \tag{4.3.1}$$

Of course we need $r \geq d$ for stability. Interestingly, when $r = d$, the cost $g^* = 0$.

Why is that so?

The (perhaps) most important formula for practical inventory management is the Economic Order Quantity (EOQ) which follows from (4.3.1) by taking $r \to \infty$ and $b \to \infty$, which amount to saying that replenishments arrive immediately and backlogging is not allowed. This results in

$$g^*_{EOQ} = \sqrt{2Kdh} \qquad s^* = 0, \qquad Q^* = \sqrt{\frac{2Kd}{h}},$$

where we write Q^* instead of S^* to indicate that we order in fixed quantities. Note that the cost in these limits *increases*. Observe also that the policy that orders the EOQ is an (s,S)-policy, a (Q,r)-policy, and a (T,S)-policy.

Because we add binding constraints to the optimization problem.

An interesting extension is to consider a system which has constraints on cycle length and order size. Can you find an optimal policy?

Recall the policies of Section 2.3.

TF 4.3.1. Consider a machine which fails regularly and needs to be repaired. Assuming the repairs take a negligible amount of time, the time between machine failures has density f_X, and each repair costs c. Claim, then renewal reward theory inform us that: cost rate $= \frac{c}{\int_0^\infty x f(x)\,\mathrm{d}x}$.

TF 4.3.2. Claim: For the $G/G/1$ queue, if $\mathsf{E}[U]$ is the expected busy time and $\mathsf{E}[I]$ is the expected idle time, then

$$\rho = \mathsf{E}[U]/(\mathsf{E}[U] + \mathsf{E}[I]).$$

TF 4.3.3. Claim. The following argument proves that $\lambda = \frac{1}{\mathbb{E}[X]}$ by means of the renewal reward theorem. Let $Y(t) = t$ and $T_k = A_k$, from this it follows that, $X_k = A_k - A_{k-1}$. Moreover, $\lim_{t\to\infty} \frac{N(t)}{t} = \lambda$. As $\lim_{t\to\infty} \frac{Y(t)}{t} = 1$ we have that $1 = \lambda\mathbb{E}[X]$ proving the result.

TF 4.3.4. An (s,S)-policy is a restocking policy where s denotes the size of replenishment orders, and S represents the speed at which items are produced.

Ex 4.3.5. Use the renewal reward theorem to relate the load $\rho = \lambda\,\mathsf{E}[S]$ to the limiting fraction of time the server of a stable $G/G/1$ queue is busy.

Ex 4.3.6. We can also derive the relation between utilization and the fraction of busy time of the server of a $G/G/1$ queue by considering the inequalities

$$\sum_{k=1}^{A(t)} S_k \geq \int_0^t \mathbb{1}_{L(s)>0}\,\mathrm{d}s \geq \sum_{k=1}^{D(t)} S_k. \tag{4.3.2}$$

Explain the above and show that by taking appropriate limits we find $\lambda\,\mathsf{E}[S] \geq \rho \geq \delta\,\mathsf{E}[S]$.

Ex 4.3.7. Show for the $G/G/1$ queue that $\rho = \mathsf{E}[U]/(\mathsf{E}[U] + \mathsf{E}[I])$ where $\mathsf{E}[U]$ is the expected busy time and $\mathsf{E}[I]$ the expected idle time. (The busy time of a server is the time between starting after an idle time until a new idle time starts.)

4.4 LITTLE'S LAW

There exists a fascinating connection, known as *Little's law*, between the average sojourn time of a job in an input-output system and the long-run time-average number of jobs within that system. To heuristically understand this relationship let's consider an illustrative example. Imagine a highway segment between points A and B spanning 100 km. Every minute, one car enters the highway, and each car maintains a constant speed of 50 km/h. Now, since each car spends 2 hours on the highway, there must be 120 cars on the highway simultaneously. The objective in this section is to demonstrate that under a few simple conditions this relation generalizes to stochastic systems, in which case it takes the form $\mathsf{E}[L] = \lambda \mathsf{E}[J]$, i.e., the average number of 'things' in a system is the rate at which these 'things' arrive times the average time they spend in the system.

100 km / 50 km/h = 2 hours

2 h × 60 min/h × 1 car/min = 120 cars.

Use the physical dimensions of the components of Little's law to check that $\mathsf{E}[J] \neq \lambda \mathsf{E}[L]$.

Define for the kth job the indicator function $I_k(s) := \mathbb{1}_{A_k \leq s < D_k}$ to say that $I_k(s) = 1$ if job k is present in the system at time s and zero otherwise. Consequently, $\int_0^t I_k(t)\,\mathrm{d}s$ represents the total time job k has spent in the system until time t, and

$$J_k = \int_0^\infty I_k(s)\,\mathrm{d}s, \qquad\qquad L(t) = \sum_{k=1}^{\infty} I_k(t),$$

are, respectively, the sojourn time of the kth job and the number of jobs in the system at time t.

Theorem 4.4.1 (Little's Law, $\mathsf{E}[L] = \lambda \mathsf{E}[J]$)**.** If $\delta = \lambda < \infty$, and $\frac{1}{n}\sum_{k=1}^{n} J_k \to \mathsf{E}[J] < \infty$ as $n \to \infty$, i.e., this limit exists and is finite, then the limit $\mathsf{E}[L] := \lim_{t\to\infty} \frac{1}{t}\int_0^t L(s)\,\mathrm{d}s$ exists, and $\mathsf{E}[L] = \lambda \mathsf{E}[J]$.

Proof. There are three ways to charge job k for the service it gets: just after departure, for the amount of service it received up to t, or right upon arrival. A comparison of the costs of these three charging schemes shows that

$$J_k \mathbb{1}_{D_k \leq t} \leq \int_0^t I_k(s)\,\mathrm{d}s \leq J_k \mathbb{1}_{A_k \leq t}.$$

Taking the sum over this inequality gives

Observe that $\sum_{k=1}^{\infty} J_k \mathbb{1}_{D_k \leq t} = \sum_{k=1}^{D(t)} J_k$ and likewise for $A(t)$.

$$\frac{1}{t}\sum_{k=1}^{D(t)} J_k \leq \frac{1}{t}\int_0^t \sum_{k=1}^{\infty} I_k(s)\,\mathrm{d}s = \frac{1}{t}\int_0^t L(s)\,\mathrm{d}s \leq \frac{1}{t}\sum_{k=1}^{A(t)} J_k.$$

By using the assumptions, for the LHS,

$$\frac{1}{t}\sum_{k=1}^{D(t)} J_k = \frac{D(t)}{t}\frac{1}{D(t)}\sum_{k=1}^{D(t)} J_k \to \delta \mathsf{E}[J];$$

likewise the RHS $\to \lambda \mathsf{E}[J]$. Next, as $\delta = \lambda < \infty$, the limits of the LHS and RHS are the same. Consequently, the limit $\frac{1}{t}\int_0^t L(s)\,\mathrm{d}s \to \mathsf{E}[L]$ exists, and $\lambda \mathsf{E}[J] = \mathsf{E}[L]$. □

The conditions are somewhat subtle: $\delta = \lambda < \infty$ does not imply that $\lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} J_k < \infty$, even if this limit exists. For instance, consider a single server queue, take $A_k = k$, and $S_k = 1$ if k is not a square but $S_k = 2$

if k is a square. Then for very large k, $D_k \sim k - \sqrt{k}$. Hence, $D_k/k \to 1$. However, for large s, $L(s) = A(s) - D(s) \approx \sqrt{s}$, so that $t^{-1}\int_0^t \sqrt{s}\,\mathrm{d}s \to \infty$ as $t \to \infty$.

TF 4.4.1. Claim: Little's law states for a rate-stable $G/G/1$ queue that $\mathsf{E}[J] = \lambda\,\mathsf{E}[L]$.

TF 4.4.2. Claim: for Little's law to be true for any input-output system the equality

$$\int_0^T L(s)\,\mathrm{d}s = \sum_{k=1}^{A(T)} W_k$$

must hold for all $T \geq 0$.

TF 4.4.3. Consider the server of the $G/G/1$ queue as a system by itself. The time jobs stay in this system is $\mathsf{E}[S]$, and jobs arrive at rate λ. Claim: It follows from Little's law that the fraction of time the server is busy is $\lambda\,\mathsf{E}[S]$.

TF 4.4.4. Suppose there are 10 jobs present at the $M/M/1$ queue with arrival rate $\lambda = 3$ and service rate $\mu = 4$ per hour. Claim: The time to clear the system follows from Little's law and is $3 \cdot 10 = 30$ hours.

Ex 4.4.5. Think of all servers of the (stable) $G/G/c$ queue as being put in a box. Jobs enter the box at rate λ and stay $\mathsf{E}[S]$ in this box. Use Little's law to conclude that the time-average number of busy server is $\mathsf{E}[L_s] = \lambda\,\mathsf{E}[S]$.

Ex 4.4.6. Let's take 'life' itself as an input-output system, and ask how many people are born and die in the Netherlands per week. For ease, assume that in 2023 the Netherlands has 16 M inhabitants, and people have an expected lifetime (i.e., the time they remain in the system called 'life') of 80 years. With Little's law we conclude that the rate λ into the system (i.e., people born) is $16M/80 = 0.2M$ per year. Again for ease, if a year has 50 weeks, there are 4000 births per week. If the size of the population remains about constant, also around 4000 people die per week.

As it turns out, the death rate in 2023 is about 2900 per week. This is much lower than the above estimate, and a more precise computation does not close the gap. Can you explain this paradoxical difference?

4.5 POISSON ARRIVALS SEE TIME AVERAGES

Suppose jobs arrive exactly at the start of an hour and require 59 minutes of service. If we sample the server occupation at job arrival times, the server is always free as seen by the arrivals, while if we track the server over time, it is nearly always busy. Thus, what jobs see upon arrival is *in general not equal* to what the server perceives. However, when jobs arrive as a Poisson process, both sampling methods produce the same number, a result known as the *Poisson arrivals see time averages* (*PASTA*) property. Here we will discuss this property in more detail, and in the remainder of the book we will use it time and again.

LET US SAY that the system is in *state n* at time t when $L(t) = n$. Then define

$$A(n,t) := \sum_{k=1}^{\infty} \mathbb{1}_{A_k \le t} \mathbb{1}_{L(A_k-)=n} = \sum_{k=1}^{A(t)} \mathbb{1}_{L(A_k-)=n} \tag{4.5.1}$$

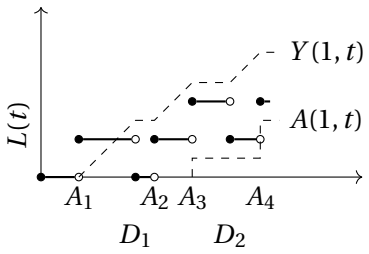
as the number of arrivals up to time t that saw the system in state n. Next, let

$$Y(n,t) := \int_0^t \mathbb{1}_{L(s)=n} \, \mathrm{d}s$$

be the total time the system spends in state n during $[0,t]$. To see the practical relevance of this definitions, suppose it costs w per unit time to have a job in the system. Then, the total cost up to time t is $w\sum_{n=0}^{\infty} nY(n,t)$.

Note the A_k-; since $L(t)$ is right-continuous, we concentrate on what an arrival *sees just before it's arrival*

[4.5.5],[4.5.6]



We subtracted $1/2$ from the graph of $A(1,t)$, for otherwise this overlaps with the graph of Y.

With these definitions, assume the following limits exist:

$$\lambda := \lim_{t\to\infty} \frac{A(t)}{t}, \qquad \lambda(n) := \lim_{t\to\infty} \frac{A(n,t)}{Y(n,t)}, \tag{4.5.2a}$$

$$p(n) := \lim_{t\to\infty} \frac{Y(n,t)}{t}, \qquad \pi(n) := \lim_{t\to\infty} \frac{1}{A(t)} \sum_{k=1}^{A(t)} \mathbb{1}_{L(A_k-)=n}. \tag{4.5.2b}$$

Here, $\lambda(n)$ has the interpretation of the arrival rate *while* the system is in state n, $p(n)$ is the long-run fraction of *time* spent in state n, and $\pi(n)$ is the long-run fraction of *arrivals* that observe n customers in the system.

Theorem 4.5.1. If all limits exist, then,

$$\lambda\pi(n) = \lambda(n)p(n). \tag{4.5.3}$$

Proof. On the one hand,

$$\lim_{t\to\infty} \frac{A(n,t)}{t} = \lim_{t\to\infty} \frac{A(n,t)}{Y(n,t)} \frac{Y(n,t)}{t} = \lambda(n)p(n).$$

On the other,

$$\lim_{t\to\infty} \frac{A(n,t)}{t} = \lim_{t\to\infty} \frac{A(n,t)}{A(t)} \frac{A(t)}{t} = \pi(n)\lambda.$$

□

This result has a simple consequence. Write $\mathscr{S}$ for all the states the system can attain. Clearly, any arrival must see the system in some state, hence $\sum_{n\in\mathscr{S}} \pi(n) = 1$. Thus, summing over n in (4.5.3), the overall arrival rate λ is such that

When there is no upper bound on L, then $\mathscr{S} = \mathbb{N}$, but when the L is bounded by some number K, then $\mathscr{S} = \{0,1,\ldots,K\}$.

$$\lambda = \sum_{n\in\mathscr{S}} \lambda(n)p(n). \tag{4.5.4}$$

With this theorem we can obtain second result of utmost importance.

Theorem 4.5.2. If the arrival rate does not depend on the state of the system, i.e., $\lambda(n) = \lambda$, the sample probabilities $\{\pi(n)\}$ are equal to the time-average probabilities $\{p(n)\}$, i.e.,

$$\lambda(n) = \lambda \iff \pi(n) = p(n). \tag{4.5.5}$$

Proof. From (4.5.3), $\pi(n) = \lambda(n)p(n)/\lambda$. By the assumption, $\lambda = \lambda(n)$, so the λ's cancel. □

As the example in the introduction of this section shows, this property is not satisfied in general. However,

A rigorous proof of this is hard, see e.g., El-Taha and Stidham Jr. [1998]

Corollary 4.5.3 (Poisson Arrivals See Time Averages (PASTA))**.** If the arrival process is Poisson, then $\lambda(n) = \lambda$, in which case $\pi(n) = p(n)$.

BY DISCRETIZING TIME we can provide a simple and intuitive way to understand when PASTA does (not) hold. If in some period n a job arrives, i.e., $a_n = 1$, and the system contains m jobs at the end of the previous period, i.e., $L_{n-1} = m$, then we would say that the arrival sees m jobs in the system upon arrival. Thus, for n large, $\pi(m) \approx \mathsf{P}\{L_{n-1} = m | a_n = 1\}$. Now, with Bayes' law:

$$\mathsf{P}\{L_{n-1} = m | a_n = 1\} = \frac{\mathsf{P}\{a_n = 1, L_{n-1} = m\}}{\mathsf{P}\{a_n = 1\}}.$$

Observing that L_{n-1} is a function of $\{a_i\}_{i=1}^{n-1}$), then, when a_n is independent of $\{a_i\}_{i=1}^{n-1}$, a_n and L_{n-1} are independent. In that case $\mathsf{P}\{a_n = 1, L_{n-1} = m\} = \mathsf{P}\{a_n = 1\}\,\mathsf{P}\{L_{n-1} = m\}$, and by canceling $\mathsf{P}\{a_n = 1\}$ in the fraction, we see that

$$\pi(m) \approx \frac{\mathsf{P}\{a_n = 1\}\,\mathsf{P}\{L_{n-1} = m\}}{\mathsf{P}\{a_n = 1\}} = \mathsf{P}\{L_{n-1} = m\} \approx p(m),$$

where the last approximation follows from the fact that $\mathsf{P}\{L_{n-1} = m\}$ is the fraction of time $L_{n-1} = m$.

For mathematically interested students: The assumed strong convergence $N^{-1}\sum_{n=1}^{N} \mathbb{1}_{L_n=m} \to p(m)$ as $N \to \infty$ implies the weak converge $\mathsf{P}\{L_n = m\} \to p(m)$ as $n \to \infty$.

However, PASTA does not hold when a_n depends on L_{n-1}. For instance, if $a_n = 0$ when $L_{n-1} > 0$, and $c_k \sim \text{Bern}(d)$ for some small probability d, then $\pi(1) = 0$, but $\mathsf{P}\{L_{n-1} = 0\} > 0$ in general.

WITH JUST PASTA, Little's law and memorylessness we can already achieve quite a few interesting results for the $M/M/1$ queue. First observe that

$$\mathsf{E}[J] \overset{1}{=} \mathsf{E}[L]\,\mathsf{E}[S] + \mathsf{E}[S] \overset{2}{=} \lambda\,\mathsf{E}[J]\,\mathsf{E}[S] + \mathsf{E}[S] \overset{=}{\rho}\,\mathsf{E}[J] + \mathsf{E}[S],$$

For the $M/G/1$ queue, things are bit subtler, see [4.5.8] and Section 5.4.

where 1 follows from PASTA, 2 from Little's law $\mathsf{E}[L] = \lambda\,\mathsf{E}[J]$, and 3 from $\rho = \lambda\,\mathsf{E}[S]$. Consequently,

$$\mathsf{E}[J] = \frac{\mathsf{E}[S]}{1-\rho}, \tag{4.5.6a}$$

and then step by step,

$$\mathsf{E}[L] = \lambda\,\mathsf{E}[J] = \frac{\lambda\,\mathsf{E}[S]}{1-\rho} = \frac{\rho}{1-\rho}, \tag{4.5.6b}$$

$$\mathsf{E}[W] = \mathsf{E}[L]\,\mathsf{E}[S] = \frac{\rho}{1-\rho}\,\mathsf{E}[S] \tag{4.5.6c}$$

$$\mathsf{E}[Q] = \lambda\,\mathsf{E}[W] = \frac{\rho^2}{1-\rho}, \tag{4.5.6d}$$

$$\mathsf{E}[L_s] = \mathsf{E}[L] - \mathsf{E}[Q] = \frac{\rho}{1-\rho} - \frac{\rho^2}{1-\rho} = \rho, \tag{4.5.6e}$$

TO LET YOU think a bit deeper about the PASTA property, and statistics in general, consider the next interesting riddle. The problem is known as the 'inspection paradox'.

Look this paradox up on the web. You should be aware of such biases in data science.

In a small Midwestern town there lived a retired railroad engineer named William Johnson. The main line on which he had worked for so many years passed through the town. Mr. Johnson suffered from insomnia and would often wake up at any odd hour of the night and be unable to fall asleep again. He found it helpful, in such cases, to take a walk along the deserted streets of the town, and his way always led him to the railroad crossing. He would stand there thoughtfully watching the track until a train thundered by through the dead of the night. The sight always cheered the old railroad man, and he would walk back home with a good chance of falling asleep.

After a while he made a curious observation; it seemed to him that most of the trains he saw at the crossing were traveling eastward, and only a few were going west. Knowing very well that this line was carrying equal numbers of eastbound and westbound trains, and that they alternated regularly, he decided at first that he must have been mistaken in this reckoning. To make sure, he got his little notebook, and began putting down "E" or "W," depending on which way the first train to pass was traveling. At the end of a week, there were five "E's" and only two "W's" and the observations of the next week gave essentially the same proportion. Could it be that he always woke up at the same hour of night, mostly before the passage of eastbound trains?

Being puzzled by this situation, he decided to undertake a rigorous statistical study of the problem, extending it also to the daytime. He asked a friend to make a long list of arbitrary times such as 9:35 a.m., 12:00 noon, 3:07 p.m., and so on, and he went to the railroad crossing punctually at these times to see which train would come first. However, the result was the same as before. Out of one hundred trains he met, about seventy-five were going east and only twenty-five west. In despair, he called the depot in the nearest big city to find whether some of the westbound trains had been rerouted through another line, but this was not the case. He was, in fact, assured that the trains were running exactly on schedule, and that equal numbers of trains daily were going each way. This mystery brought him to such despair that he became completely unable to sleep and was a very sick man.

Can you reveal the mystery thereby solving the sleeping problems of Mr. Johnson?

The story comes from the book 'Puzzle Math' by G. Gamow and M. Stern, https://www.arvindguptatoys.com/arvindgupta/puzzlemath.pdf. *The interesting solution can also be found there.*

TF 4.5.1. Claim: the PASTA property implies that $\sum_{n=0}^{\infty} np(n) = \sum_{n=0}^{\infty} n\pi(n)$ for any $G/M/1$ queue.

TF 4.5.2. Consider a stable $M/G/1$ queue. Claim: By the PASTA property, a fraction ρ of the arrivals sees the server occupied, while a fraction $1-\rho$ sees a free server.

TF 4.5.3. Claim: For the $M/G/1$ queue it follows from the PASTA property that

$$\mathsf{E}[J] = \sum_{n=0}^{\infty} \mathsf{E}[W \mid L = n]\,\pi(n) + \mathsf{E}[S].$$

TF 4.5.4. Let $L_S(s)$ be the number of items in the system at time s. Define

$$\alpha = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} L_S(A_k).$$

For the $M/G/1$ queue we can use PASTA to see that $1-\rho=\alpha$.

Ex 4.5.5. Removed.

Ex 4.5.6. If $\lambda>\delta$, can it happen that $\lim_{t\to\infty} A(n,t)/t>0$ for some (finite) n?

Ex 4.5.7. When $\lambda\neq\delta$, is $\pi(n)\geq\delta(n)$?

Ex 4.5.8. While (4.5.6c) is true for the $M/M/1$, it is *not true for the* $M/G/1$ *queue*. Why is that?

4.6 LEVEL CROSSING AND BALANCE EQUATIONS

Consider a sample path of $L:=\{L(t):t\geq 0\}$ of a queueing or inventory system in which jobs or items arrive and depart one by one. We say that the sample path *up-crosses level* n at time t when the number of jobs in the system changes from n to $n+1$ due to an arrival, in other words, when $L(t-)=n$ and $L(t)=n+1$. The sample path *down-crosses level* n at time t when, due to a departure, $L(t-)=n+1$ and $L(t)=n$. Clearly, the number of up-crossings and down-crossings cannot differ by more than 1 at any time, because it is only possible to down-cross level n after an up-crossing (or the other way around). This simple idea will prove to be a key stepping stone in the analysis of many stochastic systems.

SIMILAR TO THE definitions for $A(n,t)$ and $\lambda(n)$, let

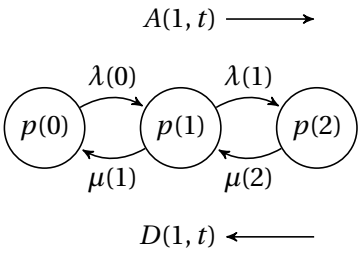
$$D(n,t):=\sum_{k=1}^{D(t)}\mathbb{1}_{L(D_k)=n},\qquad \mu(n+1):=\lim_{t\to\infty}\frac{D(n,t)}{Y(n+1,t)}, \tag{4.6.1}$$

respectively, denote the number of departures up to time t that *leave* n *customers behind* and the departure rate from state $n+1$.

Note that we write $L(D_k)$ and not $L(D_k-)$: to leave n jobs behind, the system must contain $n+1$ jobs just prior to the departure.

Theorem 4.6.1 (Level-crossing)**.** When jobs arrive and depart as single units,

$$\lambda(n)p(n)=\mu(n+1)p(n+1). \tag{4.6.2}$$



(4.6.3) *is simple, but has profound consequences.*

Proof. By the assumption, $|L(t)-L(t-)|\leq 1$ for all $t\geq 0$. But this implies that

$$|A(n,t)-D(n,t)|\leq 1. \tag{4.6.3}$$

From this observation it follows immediately that

$$\lim_{t\to\infty}\frac{A(n,t)}{t}=\lim_{t\to\infty}\frac{D(n,t)}{t}. \tag{4.6.4}$$

But

$$\lim_{t\to\infty}\frac{A(n,t)}{t}=\lim_{t\to\infty}\frac{A(n,t)}{Y(n,t)}\frac{Y(n,t)}{t}=\lambda(n)p(n),$$
$$\lim_{t\to\infty}\frac{D(n,t)}{t}=\lim_{t\to\infty}\frac{D(n,t)}{Y(n+1,t)}\frac{Y(n+1,t)}{t}=\mu(n+1)p(n+1),$$

from which the claim follows. □

SUPPOSE WE CAN specify the arrival and service rates $\lambda(n)$ and $\mu(n)$, then we can easily compute the long-run fraction of time $p(n)$ that the system contains n jobs. To see this, rewrite (4.6.2) as

In Section 5.1 and onward we will model many queueing situations by making suitable choices for $\lambda(n)$ and $\mu(n)$.

$$p(n+1) = \frac{\lambda(n)}{\mu(n+1)} p(n). \tag{4.6.5}$$

A straightaway recursion then leads to

$$p(n+1) = \frac{\lambda(n)\lambda(n-1)\cdots\lambda(0)}{\mu(n+1)\mu(n)\cdots\mu(1)} p(0).$$

Thus, $p(n)$, $n \geq 1$, is just $p(0)$ times a constant, and this constant is based on arrival and service rates.

To determine $p(0)$ we can use the fact that the numbers $p(n)$ represent probabilities. Hence, from the normalizing condition $\sum_{n=0}^{\infty} p(n) = 1$, we get $p(0) = G^{-1}$ with G being the *normalization constant*

$$G = 1 + \sum_{n=0}^{\infty} \frac{\lambda(n)\lambda(n-1)\cdots\lambda(0)}{\mu(n+1)\mu(n)\cdots\mu(1)}.$$

Finally, once we have $p(n)$, it is easy to compute the time-average number of jobs in the system and the long-run fraction of time the system contains at least n:

It is important to realize that this is not necessarily the same as what jobs see upon arrival.

$$\mathsf{E}[L] = \sum_{n=0}^{\infty} n p(n), \qquad \mathsf{P}\{L \geq n\} = \sum_{i=n}^{\infty} p(i).$$

WITH THE DEFINITIONS of $A(n,t)$ and $D(n,t)$, we can establish a nice relation between $\pi(n)$ and $\delta(n)$, i.e., statistics as obtained by the departures. Define, analogous to $\pi(n)$,

$$\delta(n) := \lim_{t\to\infty} \frac{D(n,t)}{D(t)}$$

as the long-run fraction of jobs that leave n jobs *behind*. From (4.6.4) and supposing that $A(n,t) = A(n,t)$,

$$\frac{A(t)}{t}\frac{A(n,t)}{A(t)} = \frac{A(n,t)}{t} = \frac{A(n,t)}{t} \approx \frac{D(n,t)}{t} = \frac{D(t)}{t}\frac{D(n,t)}{D(t)}.$$

Taking limits at the left and right, and using (4.1.1), we obtain for the $G/G/c$ queue

Because customers arrive and leave as single units in a (rate-stable) $G/G/c$ queue.

$$\lambda\pi(n) = \delta\delta(n). \tag{4.6.6}$$

Consequently, for the (stable) $G/G/c$ queue the statistics obtained by arrivals is the same as statistics obtained by departures, i.e.,

$$\lambda = \delta \iff \pi(n) = \delta(n). \tag{4.6.7}$$

WHEN JOBS CAN BLOCKED, we need to make a distinction between jobs that just arrive and jobs that actually enter. Jobs that are blocked do not enter, hence such arrivals do not produce up-crossings. To handle this more general case, we consider

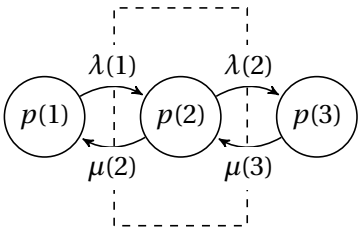
$$A^+(n,t) := \sum_{k=1}^{A(t)} \mathbb{1}_{L(A_k-)=n}\, \mathbb{1}_{L(A_k)\geq n+1}, \qquad \lambda^+(n) := \lim_{t\to\infty} \frac{A^+(n,t)}{Y(n,t)}, \tag{4.6.8}$$

so that $A^+(n,t)$ only counts arrivals that up-cross level n, and $\lambda^+(n)$ corresponds to the rate at which jobs *enter* when the system contains n jobs. We should replace $\lambda(n)$ by $\lambda^+(n)$ at all other relevant places, for instance, (4.6.2) becomes

$$\lambda^+(n)p(n) = \mu(n+1)p(n+1).$$

Observe that $\lambda(n) > \lambda^+(n)$ when a fraction of jobs is blocked, otherwise $\lambda(n) = \lambda^+(n)$.

IT IS IMPORTANT to realize that the level-crossing argument cannot always be used, as it is not always possible to split the state space into two disjoint parts by 'drawing a line' between two states. For a more general approach, we focus on a single state and count how often this state is entered and left. Specifically, define $I(n,t) = A(n-1,t) + D(n,t)$ as the number of times the queueing process enters state n either due to an arrival from state $n-1$ or due to a departure leaving n jobs behind. Similarly, $O(n,t) = A(n,t) + D(n-1,t)$ counts how often state n is left either by an arrival (to state $n+1$) or a departure (to state $n-1$).



Just like (4.6.3), it is evident that $|I(n,t) - O(n,t)| \le 1$, and this implies

$$\lambda(n-1)p(n-1) + \mu(n+1)p(n+1) = (\lambda(n) + \mu(n))p(n). \tag{4.6.9}$$

[4.6.9]

These equations hold for any $n \ge 1$ and are known as the *balance equations*. We will use these equations later in the book to analyze queueing networks.

LEVEL CROSSING ARGUMENTS CAN also be applied to continuous systems. With this we can find a closed form expression for the waiting time of the $M/G/1$ queue. However, this is a bit more complicated as the argument hinges on an integral equation. If you don't like developing mathematical skills, you may skip this derivation.

Suppose that the waiting time W of the $M/G/1$ queue has a density f, that is, the cdf $F(x) = \mathsf{P}\{W \le x\}$ has a derivative $f(x)$ for all $x > 0$. (F is only differentiable from the right at 0 because $\mathsf{P}\{W = 0\} > 0$, while $\mathsf{P}\{W \le x\} = 0$ for $x < 0$.) Consider some level $x > 0$. By PASTA, the rate of arriving jobs that see a waiting time $y \in [0,x)$ is equal to $\lambda f(y)$. (Observe that we use the splitting property of Poisson processes.) To up-cross level x from state $y < x$, the service time must be at least $x - y$, the probability of which is $G(x-y) = \mathsf{P}\{S > x-y\}$. Adding up the rates for all possible waiting times below x gives that $\lambda \int_0^x f(y)G(x-y)\,\mathrm{d}y$ is the up-crossing rate of level x. The downcrossing rate is $f(x)$ because a fraction $f(x)$ of waiting time is served per unit time at level x. Equaling these rates gives the integral equation $\lambda \int_0^x f(y)G(x-y)\,\mathrm{d}y = f(x)$. For $x = 0$, we need to consider the atom $\mathsf{P}\{W = 0\}$. In that case, $\lambda\,\mathsf{P}\{W = 0\} = f(0+)$. We can combine all this in the following level crossing equation

For a real proof we should prove first that f exists.

$$\lambda \int_0^x G(x-y)\,\mathrm{d}F(y) = f(x).$$

Using this integral equation,

$$\begin{aligned}\mathsf{E}[W] &= \int_0^\infty x f(x)\,\mathrm{d}x = \lambda \int_0^\infty x \int_0^x G(x-y)\,\mathrm{d}F(y)\,\mathrm{d}x = \\ &= \lambda \int_0^\infty \int_0^\infty x\,\mathbb{1}_{y\le x} G(x-y)\,\mathrm{d}x\,\mathrm{d}F(y) \\ &= \lambda \int_0^\infty \int_0^\infty (u+y) G(u)\,\mathrm{d}u\,\mathrm{d}F(y) \\ &= \lambda \int_0^\infty (\mathsf{E}\left[S^2\right]/2 + y\,\mathsf{E}[S])\,\mathrm{d}F(y) \\ &= \lambda\,\mathsf{E}\left[S^2\right]/2 + \lambda\,\mathsf{E}[S]\,\mathsf{E}[W].\end{aligned}$$

Bringing $\lambda\,\mathsf{E}[S]\,\mathsf{E}[W] = \rho\,\mathsf{E}[W]$ to the left, and dividing by $1-\rho$ gives

$$\mathsf{E}[W] = \lambda \frac{\mathsf{E}\left[S^2\right]}{2(1-\rho)}.$$

It is left to compute the second moment of the service time S. We will derive this formula with the renewal reward equation in Section 5.4.

As a last point of interest, let's derive the density of the waiting time for the $M/M/1$ queue. As $S \sim \mathrm{Exp}(\mu)$ in this case, f must satisfy $f(x) = \lambda \int_0^x e^{-(x-y)\mu}\,\mathrm{d}F(y)$. Multiplying both sides by $e^{\mu x}$ and defining $g(x) = e^{\mu x} f(x)$ this can be rewritten to $g(x) = \lambda \int_0^x g(y)\,\mathrm{d}F(y)$. Differentiating the LHS and RHS wrt x gives the differential equation $g'(x) = \lambda g(x)$. Hence, $g(x) = Ae^{\lambda x}$, hence, $f(x) = Ae^{-(\mu-\lambda)x}$. At $x = 0$, we have that $f(0) = \lambda p(0)$. Since $p(0) = 1-\rho$, it follows that

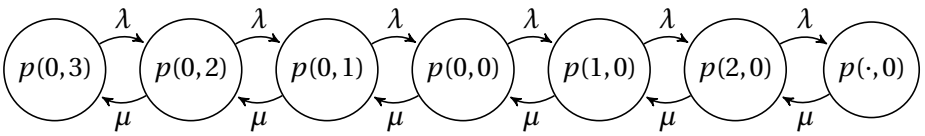
$$f(x) = \lambda(1-\rho)e^{-\mu(1-\rho)x}, \quad x > 0. \tag{4.6.10}$$

Another fun way to obtain this result is to use conditioning on the number Q of jobs found in queue at arrival, and then using that the total service of these Q customers have a gamma density. Here is the first step of the derivation:

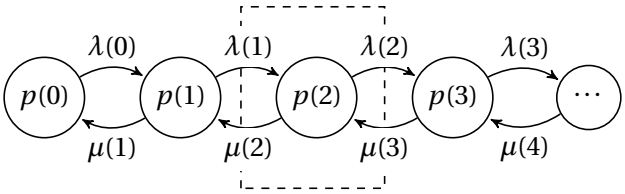
$$\begin{aligned}f(x) &= \sum_{k=1}^\infty \mathsf{P}\{W = x | Q = k\}\,\mathsf{P}\{Q = k\} \\ &= \sum_{k=1}^\infty \mathsf{P}\{S_1 + S_2 + \cdots + S_k = x\}\,\mathsf{P}\{Q = k\}.\end{aligned}$$

Substituting the Gamma distribution and some algebra leads also to (4.6.10).

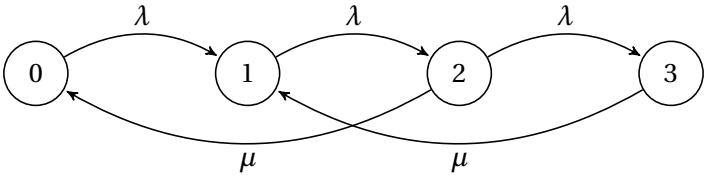
TF 4.6.1. At a large hotel, taxi cabs arrive at a rate of 15 per hour, and parties of riders arrive at the rate of 12 per hour. Whenever taxicabs are waiting, riders are served immediately upon arrival. Whenever riders are waiting, taxicabs are loaded immediately upon arrival. A maximum of three cabs can wait at a time (other cabs must go elsewhere). Let $p(i,j)$ be the steady-state probability of there being i parties of riders and j taxicabs waiting at the hotel. Claim: the transitions are modeled by the graph below.



TF 4.6.2. To obtain the *balance equations* we do not count the number of up- and down crossings of a level. Instead we count how often a box around a state, such as state 2 in the figure below, is crossed from inside and outside.



TF 4.6.3. Consider the $M^2/M^2/1/3$ queue. The graph below shows all relevant transitions.



TF 4.6.4. For the $G/G/1$ the difference between the number of out transitions' and the number of 'in transitions' is at most 1 for all t. As a consequence,

$$\begin{aligned}
\text{transitions out} &\approx \text{transitions in} \iff \\
A(n,t)+D(n-1,t) &\approx A(n-1,t)+D(n,t) \iff \\
\frac{A(n,t)+D(n-1,t)}{t} &\approx \frac{A(n-1,t)+D(n,t)}{t} \iff \\
\frac{A(n,t)}{t}+\frac{D(n-1,t)}{t} &\approx \frac{A(n-1,t)}{t}+\frac{D(n,t)}{t}.
\end{aligned}$$

Thus, under proper technical assumptions (which you can assume to be satisfied) this becomes for $t\to\infty$,

$$(\lambda(n)+\mu(n))\,p(n)=\lambda(n-1)\,p(n-1)+\mu(n+1)\,p(n+1).$$

We claim that if we specialize this result for the $M/D/1$ queue we have that $\lambda(n)=\lambda$ and $\mu(n)=\mu$, hence using PASTA,

$$(\lambda+\mu)\pi(n)=\lambda\pi(n-1)+\mu\pi(n+1).$$

Ex 4.6.5. Consider the following (silly) queueing process. At times $0,2,4,\ldots$ customers arrive, each customer requires 1 unit of service, and there is one server.

1. Find an expression for $A(n,t)$, when $L(0)=0$.
2. Show that $\pi(0)=1$ and $\pi(n)=0$, for $n>0$.
3. Check that $\lambda\pi(n)=\lambda(n)\,p(n)$.
4. Find an expression for $Y(n,t)$.
5. Compute $p(n)$ and $\lambda(n)$.

6. Compute $D(n,t)$ and $\mu(n+1)$ for $n \geq 0$.

7. Compute $\lambda(n)p(n)$ for $n \geq 0$, and check $\lambda(n)p(n) = \mu(n+1)p(n+1)$.

Ex 4.6.6. Derive $\mathsf{E}[L] = \sum_{n=0}^{\infty} np(n)$ from (4.2.4).

Ex 4.6.7. Show for the $M/G/1$ that with probability ρ a job leaves behind a busy server.

Ex 4.6.8. Consider a single server that serves one queue and serves only in batches of 2 jobs at a time (so never 1 job or more than 2 jobs). At most 3 jobs fit in the system. Single jobs arrive as a Poisson process with λ. Due to blocking, we take $\lambda(n) = \lambda$ for $n < 3$ and $\lambda(n) = 0$ for $n \geq 3$. The batch service times are exponentially distributed with mean $1/\mu$, so that by the memoryless property, $\mu(n) = \mu$.

Make a graph of the state-space and show, with arrows, the transitions that can occur and use level-crossing arguments to express the steady-state probabilities $p(n), n = 0, \ldots, 3$ in terms of λ and μ.

Ex 4.6.9. Show (4.6.9) from $|I(n,t) - O(n,t)| \leq 1$.

4.7 GRAPHICAL SUMMARIES

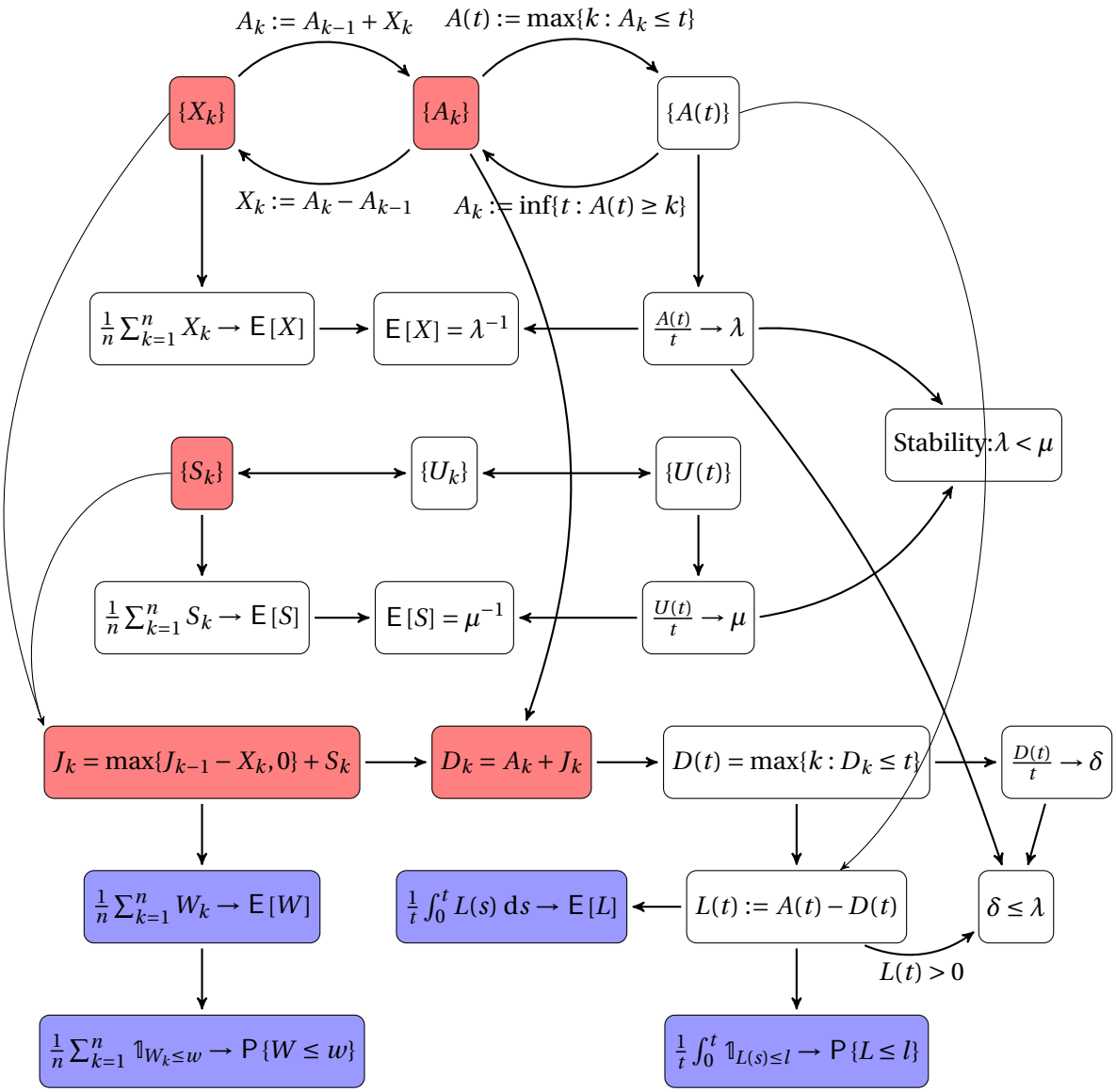


Fig. 4.7.1: An overview of the relations between the different types of times (inter-arrivals, arrivals, departures and so on) we use to construct a queueing process in continuous time.

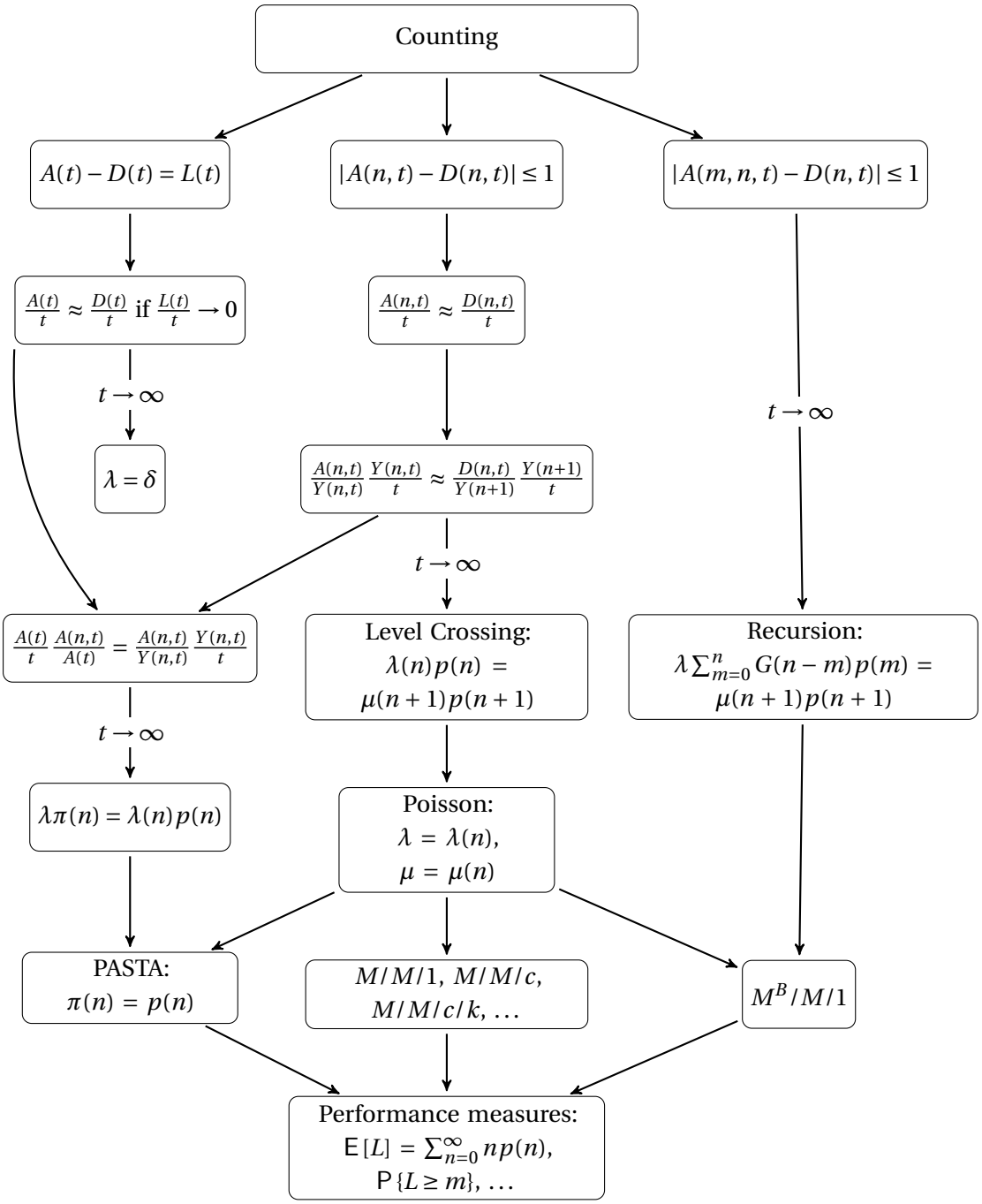


Fig. 4.7.2: An overview of the relations that derive from level-crossing arguments and the queueing systems that can be derived from these relations.

5

EXACT MODELS

In this chapter we use the concepts of Chapter 4 to model and analyze many single station queueing systems in steady state. With sample-path analysis and level-crossing, we show in Section 5.1 we consider the $M/M/1$ and simple variations. Then, in Section 5.2, we analyze a supermarket case to demonstrate how all tools come together. Section 5.3 deals with batch arrivals. In Fig. 5.4.1 we consider a simple example of a queueing system under a control rule. This leads us to a famous formula for the expectated waiting time of the $M/G/1$ queue. Finaly, Section 5.5 provides models for the control of some inventory systems.

5.1 $M/M/1$ QUEUE AND ITS VARIATIONS

In this section we develop many different queueing models by making judicious choices for $\lambda(n)$, $\lambda^+(n)$ and $\mu(n)$ for the level-crossing equations (4.6.5). We first discuss several models, then we sketch an algorithm which allows to compute the performance measures for the general case. We assume that a system that does not block arrivals in some way is stable; a systems that blocks demand is stable anyway. Note that it is much more important to understand how to specify the correct level-crossing equations than to derive closed form expressions for the performance measures.

In the models below we allow $\lambda(n)$ and $\mu(n)$ to depend on the number n in the system, but such that, given n, the inter-arrival and service times should be exponentially distributed, that is,

This very general queueing systems is sometimes written as the $M(n)/M(n)/1$ queue.

$$X|L=n \sim \text{Exp}(\lambda(n)), \qquad S|L=n \sim \text{Exp}(\mu(n)).$$

As in the general case arrivals need not enter, we consider λ^+ rather than λ in the next equations:

$$\begin{aligned}
&\mathsf{P}\{L(t+\Delta t)=n+1 \,|\, L(t)=n\} = \lambda^+(n)\Delta t + o(\Delta t), \quad n \geq 0,\\
&\mathsf{P}\{L(t+\Delta t)=n-1 \,|\, L(t)=n\} = \mu(n)\Delta t + o(\Delta t), \quad n \geq 1.
\end{aligned}$$

Below, when all arrivals are accepted $\lambda^+ = \lambda$, and then we just write $\lambda(n)$, but otherwise, we use λ^+.

THE $M/M/1$ QUEUE has a constant arrival and service rate so that we take

$$\lambda(n) = \lambda, \qquad \mu(n) = \mu.$$

With this, the level-crossing equations become

$$p(n+1) = \frac{\lambda(n)}{\mu(n+1)}p(n) = \frac{\lambda}{\mu}p(n) = \rho\, p(n), \quad \rho = \frac{\lambda}{\mu}.$$

As this holds for any $n \geq 0$, $p(n+1) = \rho^{n+1}p(0)$. Since $p(0)\sum_{n=0}^{\infty}\rho^n = p(0)/(1-\rho)$, the normalization condition $\sum_{n=0}^{\infty}p(n) = 1$ implies that $p(0) = 1-\rho$, hence

$$p(n) = (1-\rho)\rho^n, \quad n \geq 0. \tag{5.1.1}$$

It is now easy to compute the most important performance measures. The load and the utilization are equal since all jobs are accepted, hence served. With a bit of algebra, *[5.1.7]*

$$\rho = \frac{\lambda}{\mu}, \qquad \mathsf{E}[L_s] = \rho, \tag{5.1.2}$$

$$\mathsf{E}[L] = \frac{\rho}{1-\rho}, \qquad \mathsf{P}\{L > n\} = \rho^{n+1}, \tag{5.1.3}$$

recall that $\mathsf{E}[L_s]$ is the time-average number of jobs in service. Since $\mathsf{E}[L]$ is the number in the system, we see that for the number of jobs in queue,

$$\mathsf{E}[Q] = \mathsf{E}[L] - \mathsf{E}[L_s] = \frac{\rho^2}{1-\rho}.$$

THE $M/M/c$ QUEUE has c identical servers to process jobs. To model this, set

$$\lambda(n) = \lambda, \qquad \mu(n) = \mu \min\{n, c\}.$$

It is possible to derive closed form expressions for $\mathsf{E}[Q]$ and $p(n)$, but I don't find them really useful. It is much easier to use the code for the $M/M/c/K$ below, and compute the performance measures for various large values for K. When K is so large that the measures hardly change, then for all practical purposes, $\mathsf{P}\{L = K\}$ is negligible. *[5.1.8]*

Still, two measures are particularly simple:

$$\rho = \frac{\lambda}{c\mu}, \qquad \mathsf{E}[L_s] \overset{1}{=} \sum_{n=0}^{\infty} \min\{n, c\} p(n) \overset{2}{=} \frac{\lambda}{\mu},$$

where 1 shows how to compute $\mathsf{E}[L]_S$ as an expectation, and 2 follows from Little's law.

Recall, jobs spend an average time $\mathsf{E}[S]$ in service, and jobs arrive at rate λ at the servers, hence $\lambda\,\mathsf{E}[S]$ must be number of jobs at some server.

Note that $\mathsf{E}[L_s] \neq \rho$.

THE $M/M/1/K$ QUEUE blocks jobs when $L \geq K$. We can implement this by taking

$$\lambda^+(n) = \lambda \mathbb{1}_{n<K}, \qquad \mu(n) = \mu.$$

Now $\lambda(n) \neq \lambda^+(n)$.

This gives $\lambda p(n) = \mu p(n+1)$ for $n < K$, and $p(n) = 0$ for $n > K$ because $\lambda^+(n) = 0$ for $n \geq K$. Thus, in general, $\lambda^+(n)p(n) = \mu(n+1)p(n+1)$, from which follows right away that

$$\rho = \frac{\lambda}{\mu}, \qquad p(n) = \frac{1-\rho}{1-\rho^{K+1}}\rho^n.$$

There is a subtle point here to make. When the system contains K jobs, the rate at which jobs arrive at the system is $\lambda(K) = \lambda$ because jobs still *arrive* as a Poisson process, but the rate at which the jobs *enter* is $\lambda^+(K) = 0$ Consequently,

$$\lambda\pi(K) = \lambda(K)p(K) \neq \lambda^+(K)p(K) = 0.$$

Thus, since $\lambda(K) = \lambda$, we still have that $\pi(K) = p(K)$. By PASTA, the *fraction* of jobs rejected is therefore equal to $p(K)$. Moreover, since jobs are blocked, the utilization is no longer equal to the load:

$$\tilde{\rho} = \lambda(1 - p(K))/\mu = \mathsf{E}[L_s]. \tag{5.1.4}$$

THE $M/M/c/K$ QUEUE is a multiserver queue with blocking and is a immediate combination of the $M/M/c$ and $M/M/1/K$ queue:

$$\lambda^+(n) = \lambda \mathbb{1}_{n<K}, \qquad \mu(n) = \mu \min\{n, c\}.$$

THE $M/M/c/c$ QUEUE blocks jobs when all of its c servers are occupied. Thus, we take

This queueing model is also known as the Erlang B model.

$$\lambda^+(n) = \lambda \mathbb{1}_{n<c}, \qquad \mu(n) = n\mu.$$

Why is it not necessary to write $\mu(n) = \mu \min\{c, n\}$?

This queueing system is often used to determine the number of beds needed by a hospital: the beds act as servers and the patients as jobs. Given the arrival rate of patients, and the average time they occupy a bed, the problem is to find the number of beds c such that the probability $p(c)$ to block a patient is less than some threshold, 1% say. For hospitals it is reasonable to assume Poisson arrivals since patients arrive independently from a large population. Then, by PASTA, we conclude that indeed $\pi(c) = p(c)$, in other words, the blocked fraction is equal to the fraction of time $p(c)$ during which all beds are occupied. As for the service times, there are often many patients in a hospital, and they have many different reasons why they occupy a bed. Hence it is not entirely unreasonable to model the recovery times as exponentially distributed.

i.e., the expected service time

Take $\rho = \lambda/(c\mu)$. When $n < c$, $\lambda p(n) = (n+1)\mu p(n+1)$, hence

$$p(n+1) = \frac{\lambda}{(n+1)\mu} p(n) = \cdots = \frac{\lambda^{n+1}}{(n+1)!\mu^{n+1}} p(0) = \frac{(c\rho)^n}{(n+1)!} p(0).$$

The normalization constant follows as $G = \sum_{n=0}^{c} p(n)$. As there are as many servers as jobs, jobs get accepted at rate $1 - p(c)$, so that, using PASTA,

$$\mathsf{E}[Q] = 0, \qquad \mathsf{E}[L_s] = \frac{\lambda}{\mu}\big(1 - p(c)\big).$$

THE $M/M/\infty$ QUEUE is simple to analyze, as it has *ample* servers. Any arrival finds a free server, hence, its service starts directly upon arrival. Therefore,

$$\lambda(n) = \lambda, \qquad \mu(n) = n\mu.$$

By taking the limit $c \to \infty$ in the expressions of the $M/M/c$ queue we get

Or the $M/M/c/c$ queue.

$$p(n) = e^{-\lambda/\mu} \frac{(\lambda/\mu)^n}{n!}, \qquad \mathsf{E}[Q] = 0, \qquad \mathsf{E}[L] = \mathsf{E}[L_s] = \frac{\lambda}{\mu}.$$

We see that the number of busy servers in the $M/M/\infty$ queue is Poisson distributed with parameter $\lambda \mathsf{E}[S]$. We mention in passing—but do not prove it—that the same results also hold for the $M/G/\infty$ queue.

WITH A FINITE CALLING POPULATION the number of jobs is fixed, N say. This model is useful to analyze repair systems and inventory systems of spare parts. To see this, consider a factory with N machines and c mechanics. A machine can be in one of two states: working or failed. When a machine breaks down, it moves to the repair department and waits until it is repaired. When n machines are in repair, there are $N - n$ machines still working.

When $c = N$ we obtain the Ehrenfest model of diffusion, which is used to provide some insight into the second law of thermodynamics.

Thus, if λ is the rate at which a individual machine can fail, $\lambda(N-n)$ is the rate at which any of the working machines can fail, we obtain

$$\lambda(n) = \lambda(N-n), \qquad \mu(n) = \mu \min\{n, c\}.$$

Since jobs are not lost: $\lambda^+(n) = \lambda(n)$. It's easy to modify the code below, therefore it's easy to obtain numerical answers.

A CUSTOMER BALKS when the customer finds the queue too long, hence decides not to join. A simple example with customer balking is to take,

$$\lambda^+(n) = \lambda[K-n]^+, \qquad \mu(n) = \mu.$$

Balking is not necessarily the same as blocking. In the latter case, $\lambda^+(n) = \lambda \mathbb{1}_{n<K}$; in the former case, a fraction of the customers may already choose not the join the system at a lower level than K.

As the steady-state probabilities depend heavily on the form of $\lambda^+(n)$ and $\mu(n)$, we use the recursion (4.6.5) and the code below to compute the performance measures.

THE MOST GENERAL case can only be approached with numerical methods. We present example code to deal with a queue with blocking; the other models of this section are easy modifications of this. Note that we need to assume that K is finite, although it can be large. For all practical purposes, allowing K in the order of 1000 seems more than large enough.

```
import numpy as np

arrival_rate = 23
service_rate = 6
c, K = 4, 20


def labda_tilde(n):
    return arrival_rate * (K > n)


def mu(n):
    return service_rate * min(n, c)


p = np.ones(K + 1, dtype=float)
for n in range(K):
    p[n + 1] = p[n] * labda_tilde(n) / mu(n + 1)
p /= p.sum()  # normalize

Labda = sum(labda_tilde(n) * p[n] for n in range(len(p)))
print(Labda, service_rate * c) # check
EQ = sum(p[n] * max(n - c, 0) for n in range(len(p)))
ELs = sum(p[n] * min(n, c) for n in range(len(p)))
EL = sum(p[n] * n for n in range(len(p)))
print(EL, EQ + ELs) # check
```

TF 5.1.1. In the level-crossing analysis of the $M(n)/M(n)/1$ queue we claim it is necessary that the interarrival times of jobs are iid.

TF 5.1.2. For the $M/M/1$ queue, the following reasoning leads to the expected number of jobs in the system.

$$M_L(s) = \mathsf{E}\left[e^{sL}\right] = \sum_{n=0}^{\infty} e^{sn} p(n) = (1-\rho)\sum_{n} e^{sn}\rho^n$$
$$= \frac{1-\rho}{1-e^s\rho},$$

where we assume that s is such that $e^s\rho < 1$. Then,

$$M_L'(s) = (1-\rho)\frac{1}{(1-e^s\rho)^2}e^s\rho.$$

Claim: the above implies $\mathsf{E}[L] = M_L'(0) = \rho/(1-\rho)$.

TF 5.1.3. Claim: To model the $M/M/c/c+K$ queue as an $M(n)/M(n)/1$ queue we need to take $\lambda(n) = \lambda$ for all n.

TF 5.1.4. Claim: For the $M/M/c$ queue, $\mathsf{E}[Q] = \sum_{n=0}^{\infty} \max\{n-c,0\}p(n)$, wheren $p(n) = \mathsf{P}\{L = n\}$.

Ex 5.1.5. Explain that for the $M/M/1$ queue $\mathsf{E}[Q] = \sum_{n=1}^{\infty}(n-1)\pi(n)$ and use this to find that $\mathsf{E}[Q] = \rho^2/(1-\rho)$.

Ex 5.1.6. Derive the steady state probabilities $p(n)$ for a queue with a finite calling population with N jobs and N servers. What happens if $N \to \infty$?

Ex 5.1.7. Use moment-generating functions to derive $\mathsf{E}[L]$ and $\mathsf{V}[L]$ for the $M/M/1$ queue, and show that $\mathsf{P}\{L > n\} = \rho^{n+1}$.

The remaining exercises ask you to derive the expressions for the various models. You can skip them if you don't like algebra.

Th 5.1.8. Derive the following expressions for the $M/M/c$ queue:

$$\rho = \frac{\lambda}{c\mu}, \tag{5.1.5a}$$

$$p(n) = \frac{1}{G}\frac{1}{\prod_{k=1}^{n}\min\{c,k\}}(c\rho)^n, \quad n \geq 1, \tag{5.1.5b}$$

$$G = \frac{1}{p(0)} = \sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{(1-\rho)c!}, \tag{5.1.5c}$$

$$\mathsf{E}[Q] = \sum_{n=c}^{\infty}(n-c)p(n) = \frac{(c\rho)^c}{c!G}\frac{\rho}{(1-\rho)^2} \tag{5.1.5d}$$

$$\tag{5.1.5e}$$

Th 5.1.9. Check that the performance measures of the $M/M/c$ queue reduce to those of the $M/M/1$ queue if $c = 1$.

Th 5.1.10. Show that (5.1.4) is true.

Th 5.1.11. Show that as $K \to \infty$, the performance measures of the $M/M/1/K$ converge to those of the $M/M/1$ queue.

5.2 APPLICATIONS OF LEVEL-CROSSING, LITTLE'S LAW AND PASTA

In this section we apply the tools developed above to a number of interesting queueing systems. Specifically, we discuss in detail how to plan the number of cashiers at a supermarket such that the average queue length remains small. Note that in many queueing systems, the manager should balance the cost of the waiting time of customers against the cost of idle servers; recall in queueing systems always somebody has to wait, either a customer in queue or a server being idle.

LET US NEXT discuss the example of cashier planning at a supermarket. Here we keep the models simple, but we include all necessary steps to solve the a realistic planning problem. Note that many service systems, e.g. hospitals, deal with similar personnel planning problems to cover stochastic demand.

We start with specifying the *arrival process*. It is reasonable to model it as a Poisson process, as there are many potential customers, each choosing with a small probability to go to the supermarket at a certain moment in time. Thus, we only have to characterize the arrival rate. Estimating this for a supermarket is easy because the cash registers track all customers payments. This provides us with the departure time of each customer, hence we can use this to estimate the average number of arrivals per hour.

It is common to use a *demand profile* which shows the average number of customers arriving per hour. Then we model the arrivals as a Poisson process with an arrival rate that is constant during a certain hour as specified by the demand profile.

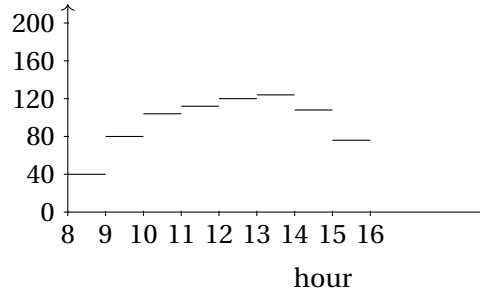


Fig. 5.2.1: A demand profile.

It is also easy to find the *service distribution* from the cash registers. The first item scanned after a payment determines the start of a service, and the payment closes the service. For ease, we model the service time distribution as exponential with mean 1.5 minutes.

For the *service objective* we prefer to keep the average number of people in the system such that $\mathsf{E}[L] \le 5$. (Other KPIs are also possible to use.)

Let us model the situation as an $M/M/1$ queue with a fast server, so that it is easy to find the smallest number of servers necessary to prevent the queue from exploding. Since $\mathsf{E}[L] = \rho/(1-\rho)$, and $\rho = \lambda \mathsf{E}[S]/c$, solving for c in the inequality $\mathsf{E}[L] \le 5$ gives

$$c \ge \frac{6}{5}\lambda \mathsf{E}[S] \approx 0.03\lambda,$$

where, with the above estimate, $\mathsf{E}[S] = 1.5/60$ hour. It's easy to apply this formula. For instance, we see in the demand profile Fig. 5.2.1 that $\lambda = 120$ customers per hour between 12 and 13, hence $c \approx 3.8 = 4$.

With this we can find a formula to convert the demand profile into a *load profile* to specify the minimal number of servers per hour needed to meet the service objective.

THE LAST STEP in the planning process is to *cover the load profile with service shifts*. This is typically not easy since shifts have to satisfy all kinds of restrictions with respect to breaks and durations. Moreover, shifts can

also have different costs: evening shifts are typically more expensive per hour.

The usual way to solve such covering problems is by means of an integer problem. For instance, suppose only 4 *shift plans* are available as shown at the right.

1. $++-++$
2. $+++-+$
3. $++-+++$
4. $+++-++$,

Shift plans. A + indicates a working hour and − a break of an hour.

For the optimization, let x_i be the number of shifts of type i and c_i the cost of this type. Then the problem is to solve $\min \sum_i c_i x_i$, such that for all hours t the shifts cover the load, i.e., $\sum_i x_i \mathbb{1}_{t \in s_i} \geq 0.03\lambda_t$. (We write $t \in s_i$ if hour t is covered by shift type i.)

FINALLY, WE NEED to address the quality of our approximation: to simplify we used the $M/M/1$ queue with a fast server, while we know that in a supermarket there are often multiple parallel cashiers. Specifically, we are interested in the ratio $\mathsf{E}[L(M/M/c]$ and $\mathsf{E}[L(M/M/1)]$ where in the $M/M/1$ queue the server works at rate c.

To understand this better, we consider a numerical example. Suppose that we have an $M/M/3$ queue with $\lambda = 5$ per day and $\mu = 2$ per day per server, and we compare $\mathsf{E}[L]$ of this queue to that of an $M/M/1$ queue with the same arrival rate but with a service rate of $\mu = 3 \cdot 2 = 6$.

Fig. 5.2.2 shows the ratio of $\mathsf{E}[L]$ for both queues as a function of ρ. Clearly, when ρ is small, this ratio is about 3. This is reasonable, because the service time in the fast $M/M/1$ is 3 times as small as the service time in the $M/M/3$ queue. Thus, when ρ is small, the time in queue is small, hence $\mathsf{E}[J] \approx \mathsf{E}[S]$. However, when ρ is large, $\mathsf{E}[J] \gg \mathsf{E}[S]$, so that in this case, the ratio between the queue lengths becomes approximately 1.

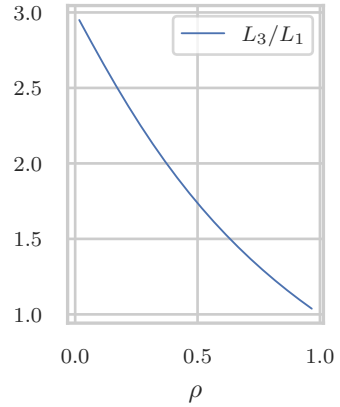


Fig. 5.2.2: The ratio of $\mathsf{E}[L]$ for the $M/M/3$ and $M/M/1$ queue.

In our supermarket model, $\rho = 5/6$, hence the approximation is not too bad.

Here is the code I used to make Fig. 5.2.2.

```
from math import factorial
import numpy as np
import matplotlib.pylab as plt

from latex_figures import fig_in_latex_format


def EL(labda, mu, c):
    rho = labda / mu / c
    G = sum((c * rho) ** n / factorial(n) for n in range(c))
    G += (c * rho) ** c / ((1 - rho) * factorial(c))
    Q = (c * rho) ** c / (factorial(c) * G) * rho / (1 - rho) ** 2
    L = Q + labda / mu
    return L


mu, c = 2.0, 3

L, load = [], []

for labda in np.linspace(0.1, 5.8, 20):
    L.append(EL(labda, mu, c) / EL(labda, c * mu, 1))
    load.append(labda / mu / c)


```

```
def cm_to_inch(cm):
    return cm / 2.54


plt.figure(figsize=(cm_to_inch(5), cm_to_inch(6)))
plt.plot(load, L, label=r"$L_3/L_1$", lw=0.7)
plt.xlabel("$\\rho$")
plt.legend()
plt.tight_layout()
plt.savefig("../figures/mm3_vs_mm1.pdf")
```

TF 5.2.1. Suppose $\frac{\lambda}{\mu} = 2$ then we claim that 3 is the minimum number of servers required to make the system stable.

TF 5.2.2. Claim: It is always better to have a single fast server than multiple slow servers which together have the same rate.

TF 5.2.3. A repair/maintenance facility would like to determine how many employees should be working in its tool crib. The service time is exponential, with mean 4 minutes, and customers arrive as a Poisson process with rate 28 per hour. With one employee we claim that the system is not rate stable.

TF 5.2.4. Consider an $M/M/c/K$ queue that is not rate stable, but jobs sometimes balk. Claim: $\lim_{t\to\infty} \mathbb{P}(L(t) = K) = 1$.

THE NEXT EXERCISES REQUIRE quite some modeling skills. Hence, they may be quite hard even thought the formulas are simple.

Ex 5.2.5. In the town hall of the municipality, people can, for example, renew their driver's license or passport. For ease, assume there is just one employee, and that the arrivals of customers are not planned. Why is it reasonable to model this queueing system as a $M/M/1$ queue?

Measurements show that $\mathsf{E}[W] = 6$ minute, and $\lambda = 5$ per minute. What are the service rate and utilization? Calculate $\mathsf{E}[Q]$, $\mathsf{E}[L]$.

There should be a sufficient number of seatings so that all waiting customers can sit down at least 90 percent of the time. How many seatings should there minimally be? What would you advice to the municipality?

Ex 5.2.6. The smallest barber shop in town works alone and has two chairs for waiting customers. Six jobs arrive per hour, and the barber needs 12 minutes for a hair cut. We model this, for ease, as an $M/M/1/3$ queue. Estimate $\mathsf{E}[Q]$ and the average number of customers served per hour. Then, estimate $\mathsf{E}[Q]$ when the barber hires another location that allows for 3 extra chairs. Quantify the effect on the number of customers served per hour.

Ex 5.2.7. Occasionally there is pizza stand at the corner of the Verlengde Visserstraat and the Westersingel in Groningen. As people (tend to) behave like sheep, the rate at which people join the queue depends on the actual queue length. When there is nobody waiting, people are somewhat hesitant to order: 'When there are no people waiting, most probably the pizzas are not good.'. However, when the queue is long, people prefer to go elsewhere.

Measurements show that the number of arrivals per hour is well captured by the array `labda = np.array([10.0, 10.0, 20.0, 20.0, 5.0])`. The pizza oven can contain two pizzas, and it takes 2 minutes to bake a pizza. We model the system as an $M/M/2$ queue.

Calculate the steady state probabilities, and from this the average arrival rate. Then, find $\mathsf{E}[L]$, $\mathsf{E}[Q]$, $\mathsf{E}[J]$ and $\mathsf{E}[W]$.

Ex 5.2.8. Fast food restaurants tend to fry pattys before customers arrive. Like this they can serve customers faster, because when a customer arrives while there is still an inventory of prepared pattys, the server only has to flip a patty in a bun, while otherwise the customer has to wait for the additional frying time of the patty. Assume the restaurant keeps at most 3 pattys on stock. To keep the model simple, assume the restaurant has just one pan to fry a patty and the frying time $S \sim \text{Exp}(\mu)$ with $\mu = 15$ per hour. Customers arrive as a Poisson process with rate $\lambda = 12$ per hour. When there is prepared patty upon arrival, the customer is served immediately, otherwise the customer has to wait for a patty.

Explain how to model this system as an $M/M/1$ system. Then compute the average number of pattys on stock, the average number of people waiting, and the average waiting time of a patty before it ends up in a bun, and the average time a person has to wait for a hamburger.

You should remember this problem. It is of fundamental importance in inventory control and known as an *inventory system with backlogging*; it is (implicitly) used by nearly any company that keeps inventory.

Ex 5.2.9 (Continuation of [5.2.8])**.** What would be the impact of allowing four pattys on stock?

Ex 5.2.10 (Continuation of [5.2.8])**.** Suppose the restaurant does not keep pattys on stock. What is the expected waiting time for customer?

5.3 $M^B/M/1$ QUEUE AND REJECTION POLICIES

Sometimes jobs arrive in batches, rather than as single units. For instance, when a car or a bus arrives at a fast-food restaurant, a batch consists of the number of people in the vehicle. When the batches arrive as a Poisson process and the individual items within a batch have exponential service times we use the shorthand $M^X/M/1$ for this queueing model. Using the renewal-reward theorem we derive expressions for the load, the expected waiting time $\mathsf{E}[W]$ in queue and expected number of jobs $\mathsf{E}[L]$ in the system.

To compute more difficult performance measures such as the loss probability $\mathsf{P}\{L > n\}$, we need expressions for the stationary distribution $\pi(n)$. We use level-crossing to derive a recursive scheme, which, when combined with a policy to reject batches, can be used to compute these probabilities.

ASSUME THAT JOBS arrive as a Poisson process with rate λ and each *job* contains multiple *items*. Denote by B_k the number of items, i.e., the batch size, of the kth job. We assume that $\{B_k\}$ is a sequence of iid discrete rvs

with common rv B. We write $S_{k,i}$ for the service time of the ith item of batch k, and assume that $\{S_{k,i}\}$ is a sequence of iid rvs with common rv S, and that the service times of the items are independent of the sizes of the batches. The utilization is therefore $\rho = \lambda \mathsf{E}[B]\,\mathsf{E}[S]$; As always we require that $\rho < 1$.

An arriving job joins the end of the queue (if present), and once the queue in front of it has been cleared, it moves in its entirety to the server. Thus, all items in one batch spend the same time in queue. Once the batch moves to the server, the server processes the items one after another until the batch is finished. Write Q^B for the number of batches in queue and L_s^B for the number of items (if any) at the server.

This is the same batching model as in Section 6.2.

Observe that the average time $\mathsf{E}\left[W^B\right]$ a batch spends in queue is

First the items at the server have to be served, then the ones in queue.

$$\mathsf{E}\left[W^B\right] = \mathsf{E}\left[L_s^B\right]\mathsf{E}[S] + \mathsf{E}\left[Q^B\right]\mathsf{E}[B]\,\mathsf{E}[S].$$

But since $\mathsf{E}\left[L_s^B\right] + \mathsf{E}\left[Q^B\right]\mathsf{E}[B] = \mathsf{E}[L]$,

$$\mathsf{E}\left[W^B\right] = \mathsf{E}[L]\,\mathsf{E}[S] = \mathsf{E}[W],$$

where $\mathsf{E}[W]$ is the average time an *item* spends in queue. We conclude that the average time a job spends in queue is the same as the average time an item spends in queue.

Substituting the equality $\mathsf{E}\left[Q^B\right] = \lambda\,\mathsf{E}\left[W^B\right]$ in the above yields

Little's law

$$\mathsf{E}[W] = \mathsf{E}\left[W^B\right] = \frac{\mathsf{E}\left[L_s^B\right]}{1-\rho}\mathsf{E}[S]. \tag{5.3.1}$$

It remains to find an expression for $\mathsf{E}\left[L_s^B\right]$.

For this we can use the renewal reward theorem. With the notation for the proof of Little's law as inspiration, define $Y(t) := \int_0^t L_s^B(s)\,\mathrm{d}s$, so that $Y(t)/t$ is the time-average number of items at the *server*. By taking D_k as the departure time of the kth batch, let $X_k = Y(D_k) - Y(D_{k-1})$. Suppose the kth job has batch size B_k, then

$$X_k|B_k = \int_{D_{k-1}}^{D_k} L_s^B(s)\,\mathrm{d}s = B_k S_{k,1} + (B_k - 1)S_{k,2} + \cdots + S_{B_k,B_k}.$$

Using Adam's law, and that the $S_{k,i}$ are iid,

$\mathsf{E}[\mathsf{E}[Y|X]] = \mathsf{E}[Y]$.

$$\mathsf{E}[X_k|B_k] = B_k\mathsf{E}[S] + (B_k-1)\mathsf{E}[S] + \cdots \mathsf{E}[S] = B_k(B_k+1)/2 \cdot \mathsf{E}[S],$$
$$\mathsf{E}[X_k] = \mathsf{E}[\mathsf{E}[X_k|B_k]] = \mathsf{E}\left[\frac{B(B+1)}{2}\right]\mathsf{E}[S] = \frac{\mathsf{E}\left[B^2\right] + \mathsf{E}[B]}{2}\mathsf{E}[S],$$

as $B_k \sim B$, i.e., distributed as the common rv B. Since $Y = \lim_{t\to\infty} Y(t)/t = \mathsf{E}\left[L_s^B\right]$, $Y = \lambda X$, $\lambda = \delta$, and $\rho = \lambda\,\mathsf{E}[B]\,\mathsf{E}[S]$, we obtain

$$\mathsf{E}\left[L_s^B\right] = \lambda\frac{\mathsf{E}\left[B^2\right]}{2}\mathsf{E}[S] + \frac{\rho}{2}. \tag{5.3.2}$$

We can brush this up by realizing that

$$\frac{\mathsf{E}\left[B^2\right]}{(\mathsf{E}[B])^2} = \frac{\mathsf{E}\left[B^2\right] - (\mathsf{E}[B])^2 + (\mathsf{E}[B])^2}{(\mathsf{E}[B])^2} = \frac{\mathsf{V}[B] + (\mathsf{E}[B])^2}{(\mathsf{E}[B])^2} = C_B^2 + 1,$$

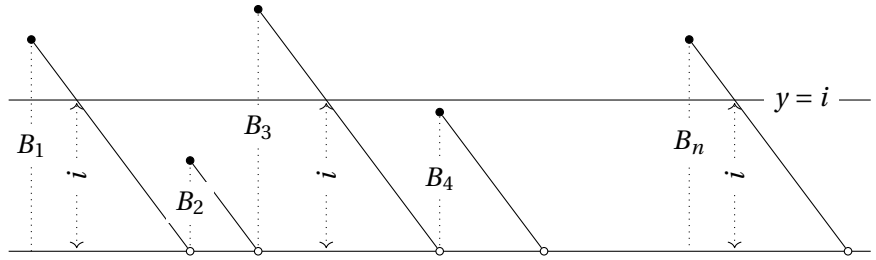


FIG. 5.3.1: *A batch crosses the line $y = i$ iff it contains at least i items. Thus, during the service of a batch with i or more items, there is precisely one service of an i-th item.*

where where $C_B^2 = \mathsf{V}[B]/(\mathsf{E}[B])^2$ is the scv of the batch size distribution. Substituting this into the above gives the final result

$$\mathsf{E}[W] = \frac{1+C_B^2}{2}\frac{\rho}{1-\rho}\mathsf{E}[B]\,\mathsf{E}[S] + \frac{1}{2}\frac{\rho}{1-\rho}\mathsf{E}[S]. \tag{5.3.3}$$

This formula is an important result. It tells us that the load increases when the load increases or the variability C_B^2 of the batch size increases, but also when $\mathsf{E}[B]\,\mathsf{E}[S]$ increases (even when the utilization ρ remains the same). As such, it gives us the most important clues about what we should try to change if we want to improve a queueing system.

RATHER THAN USING the time an entire batch spends at the server, as in the above use of the renewal-reward equation, we can concentrate on the expected time spent by single items at the server. This will provide us with a simple expression for $\mathsf{P}\{L_s = i\}$, from which we can also derive (5.3.2).

If $L_s(t)$ is the number of items (of the batch in service) at the server at time t, then $Y_i(t) = \int_0^t \mathbb{1}_{L_s(s)=i}\,\mathrm{d}s$ is the total time there are i items at the server. By sampling $Y(\cdot)$ at departure times $\{D_k\}$, we see that

$$X_k = Y_i(D_k) - Y_i(D_{k-1}) = S_{k,i}\,\mathbb{1}_{B_k \geq i},$$

where $S_{k,i}$ is the service time of the ith remaining item of the batch. Since the $\{S_{k,i}\}$ are iid with $\mathsf{E}[S_{k,i}] = \mathsf{E}[S]$, and the $\{B_k\}$ are iid and independent of the service times of the items, we obtain

$$X = \lim_{n\to\infty}\frac{1}{n}\sum_{k=1}^{n} S_{k,i}\,\mathbb{1}_{B_k\geq i} = \mathsf{E}[S\,\mathbb{1}_{B\geq i}] = \mathsf{E}[S]\,G(i-1),$$

since $G(i-1) = \mathsf{E}[\mathbb{1}_{B\geq i}] = \mathsf{P}\{B > i-1\}$ is the survivor function of B. Since, by construction, $Y_i(t)/t \to \mathsf{P}\{L_s = i\}$, we use the renewal reward theorem and rate stability, $\delta = \lambda$, to conclude that

Recall: $\mathsf{P}\{A\} := \mathsf{E}[\mathbb{1}_A]$.

$$\mathsf{P}\{L_s = i\} = \lambda\,\mathsf{E}[S]\,G(i-1) = \rho\frac{G(i-1)}{\mathsf{E}[B]},$$
$$\mathsf{E}[L_s] = \sum_{i=0}^{\infty} i\,\mathsf{P}\{L_s = i\} = \frac{\rho}{\mathsf{E}[B]}\sum_{i=1}^{\infty} iG(i-1).$$

Simplifying the summation yields (5.3.3). *[5.3.11]*

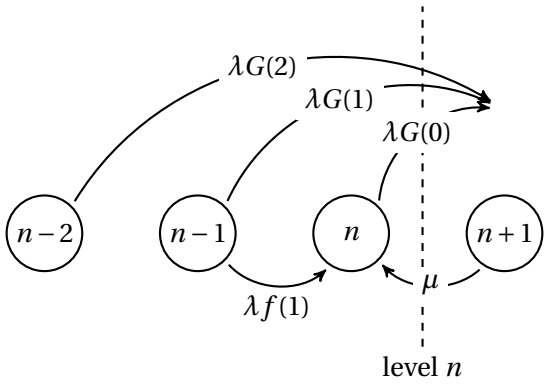


FIG. 5.3.2: *When $L(t) = n$, the arrival of any batch crosses level n from below, but when $L(t) = n-1$, only a batch larger than 1 ensures a crossing of level n, and so on.*

THE SECOND TASK is to use level-crossing arguments to find a recursive formula for the state probabilities $\{\pi(n)\}$. For this we need to generalize our earlier level-crossing equation $\lambda p(n) = \mu p(n-1)$ because, at the arrival of a batch containing many items, multiple levels will be up-crossed, see Fig. 5.3.2.

Note: n means the number of items in the system, not the number of batches.

The down-crossing rateof level n is easy: since items are still served one-by-one, we have $\mu\pi(n+1)$.

For the upcrossings of level n we should consider batch arrivals that see $m \le n$ items in the system. Since jobs arrive at rate λ, PASTA implies that $\lambda\pi(m)$ is the arrival rate of jobs that see m upon arrival; in other words, the process is thinned with a fraction $\pi(m)$. Next, when an arriving job sees m in the system, and level n has to be up-crossed, the batch size of this job must contain more than $n-m$ items. The rate at which such jobs arrive must be $G(n-m)\pi(m)\lambda$ as $G(n-m)$ is the probability that the job size is larger than $n-m$. Noting that the batch sizes are independent of the arrival times, we obtain yet again a Poisson process, but with rate $\lambda\pi(m)G(n-m)$. Finally, to the compute rate at which level n is upcrossed, we merge all these thinned Poisson processes into a new Poisson process with rate $\lambda\sum_{m=0}^{n}\pi(m)G(n-m)$.

As by level-crossing, the up-crossing and down-crossing rates must match,

$$\lambda \sum_{m=0}^{n} \pi(m)G(n-m) = \mu\pi(n+1). \tag{5.3.4}$$

It is an important check to retrieve (5.3.3) from the expression $\mathsf{E}[L] = \sum_n n\pi(n)$ and Little's law. *[5.3.12]*

IT IS LEFT to find the normalization constant. This, however, is not possible in general because (5.3.4) does not lead to a closed form expression for $\pi(n)$. A simple and sensible way out is to circumvent the problem altogether by blocking demand above some level M. There are three common policies to decide which items in a batch to accept.

Compare (5.1.1).

1. Complete acceptance: accept all batches that arrive when the system contains K or fewer items, and reject the entire batch otherwise. *[5.3.8]*

2. Partial acceptance: accept whatever fits of a batch, and reject the rest. *[5.3.7]*

3. Complete rejection: if a batch does not fit entirely into the system, it will be rejected completely. *[5.3.9]*

In the exercises we will derive recursions similar to (5.3.4) by which we can obtain $\{\pi(n)\}$. Since the recursions stop after some level M, there is a finite number of recusions, hence we can solve them numerically without much problems.

TF 5.3.1. Claim: Little's Law informs us that in the $M^B/M/1$ queue, the expected number of items in queue satisfies $\mathsf{E}[Q] = \lambda \mathsf{E}[W]$.

TF 5.3.2. For an $M^B/M/1$ queue we know that:

$$\mathsf{E}[W] = \frac{1+C_s^2}{2}\frac{\rho}{1-\rho}\mathsf{E}[B]\,\mathsf{E}[S] + \frac{1}{2}\frac{\rho}{1-\rho}\mathsf{E}[S].$$

Suppose that $\mathsf{V}[B] = 0$. Claim: this formula implies that if the batch size increases, but $\mathsf{E}[S]$ decreases such that ρ remains the same, the expected waiting time becomes smaller.

TF 5.3.3. Suppose the time to process an item has an exponential distribution. Claim: the interarrival times at which individual items arrive at an $M^B/M/1$ queue are exponentially distributed.

TF 5.3.4. Claim: this code correctly implements an $M^B/M/1$ queue with complete rejection.

```
def complete_rejection(K):
    pi, n = {}, 0
    pi[0] = 1
    while n < K:
        pi[n + 1] = sum(
            pi[m] * (S.sf(n - m) - S.sf(K - m)) for m in range(n)
        )
        pi[n + 1] *= rho
        n += 1
    return RV(pi)
```

Ex 5.3.5. A company operates a machine that receives batches of various sizes. Management likes to know how a reduction of the variability of the batch sizes would affect the average queueing time. Suppose, for the sake of an example, that the batch size $\mathsf{P}\{B=1\} = \mathsf{P}\{B=2\} = \mathsf{P}\{B=3\} = 1/3$. Batches arrive at rate $\lambda = 1/h$. The average processing time for an item is 25 minutes. Compute by how much $\mathsf{E}[L]$ would decrease if $B \equiv 2$.

Ex 5.3.6. Compute $\mathsf{E}[L]$ for the $M^B/M/1$ queue when B is geometrically distributed with $\mathsf{E}[B] = 1/p$. Check that if $\mathsf{E}[B] = 1$ the $M/M/1$ queue results.

Ex 5.3.7. Derive a set of recursions for the $M^B/M/1/K$ queue with partial acceptance.

Ex 5.3.8. Derive a set of recursions analogous to (5.3.4) to compute $\pi(n)$ for the $M^B/M/1/K$ queue with complete acceptance.

Ex 5.3.9. Derive a set of recursions for the $M^B/M/1/K$ queue with complete rejection.

Th 5.3.10. For a nonnegative discrete random variable X, use indicator functions to prove that

1. $\mathsf{E}[X] = \sum_{k=0}^{\infty} G(k)$,
2. $\sum_{i=0}^{\infty} iG(i) = \mathsf{E}\left[X^2\right]/2 - \mathsf{E}[X]/2$.

Th 5.3.11. Show that $\sum_{i=1}^{\infty} iG(i-1) = (\mathsf{E}\left[B^2\right] + \mathsf{E}[B])/2$.

Th 5.3.12. Use (5.3.4) in $\mathsf{E}[L] = \sum_{n=0}^{\infty} n\pi(n)$ to derive (5.3.3).

This is an important check both on (5.3.3) and (5.3.4).

5.4 $M/G/1$ QUEUE AND N-POLICIES

When the service times are not really well approximated by the exponential distribution, but arrivals are still Poisson and there is one server, the $M/G/1$ queue is a suitable model. In this section we will study this queueing process when subjected to a control policy and obtain as corollaries two fundamental results: the Economic Production Quantity (EPQ) formula for the optimal order quantity in an inventory system and the Pollaczek-Khinchine (PK) formula for the average waiting time of an $M/G/1$ queue. Secondly, we use sample path and the renewal reward arguments to rederive the PK formula.

IN THE QUEUEING systems we analyzed up to now, the server is always present to start serving jobs at the moment they arrive. However, in cases in which there is a cost associated with changing the server from idle to busy, this condition is typically not satisfied. For instance, the cost to heat up an oven after being idle can be quite significant; in other cases, the operator of a machine has to move from one machine to another, which takes time.

To reduce the average cost, we can use an N-policy, which works as follows. As soon as the system becomes empty, the server switches off. Then it waits until N jobs have arrived, and then it switches on. The server processes jobs until the system is empty again, then switches off, and remains idle until N new jobs have arrived, and so on, cycle after cycle.

[2.2.4]

and the server idle

or N or more items in case of batch arrivals

Note that under an N-policy, even though the load remains the same, the server has longer busy and idle times. In fact, some type of servers might use such policies to their advantage. At hospitals, for instance, doctors might prefer to let patients wait a bit until the waiting room is quite full. Like this, the doctor (server) does not have to wait for short times for late patients, but instead can collect idle times into one long stretch, and do something (they find more) useful instead.

Suppose it costs K Euro to set up the server, independent of the time it has been idle, and each job charges h Euros per unit time while in the system. Then it makes sense to first build up a queue of N jobs right after the server becomes idle, and after some time switch on the server to process

To reduce time-average setup cost.

jobs until the system is empty again. The problem is to find a switching threshold N^* that minimizes long-run average cost.

To reduce time-average queueing costs of jobs.

In this section we solve this problem for the $M/G/1$ queue, and, in passing, we obtain yet another way to compute the time-average number $\mathsf{E}[L]$ of jobs in the system. Throughout we consider the queueing process at moments at which services start.

AS A FIRST STEP, we concentrate on the expected time $T(q)$ it takes to clear the system when the server *starts working* on a job and there are q jobs in the system. We present three different ideas to obtain $T(q)$.

The first heuristic idea is this. We know that jobs arrive at rate λ and they are served at rate $\mu > \lambda$, where $\mu = 1/\mathsf{E}[S]$. Clearly, the net 'drain rate' of the queue is $\mu - \lambda$; hence we guess that

By analogy: when you have a 'queue' of q km to cycle, and your speed is v km/h, then it takes q/v h to complete the trip.

$$T(q) = \frac{q}{\mu - \lambda} = \frac{\mathsf{E}[S]}{1 - \lambda\,\mathsf{E}[S]}\,q = \frac{\mathsf{E}[S]}{1-\rho}\,q. \tag{5.4.1}$$

Observe, however, that this reasoning completely ignores the stochasticity in arrival times and services, hence, we have no guarantee that the result is correct.

For the second idea, consider a regular $M/G/1$ queue for the moment. When a job arrives to an empty system, it takes a busy time $\mathsf{E}[U]$ to get rid of this job and all jobs that arrive during the service of this first job. In other words, it takes $\mathsf{E}[U]$ time on average to move from 1 job in the system to 0, i.e., one less, jobs. But then, when there are q jobs in the system, it must take $T(q) = q\,\mathsf{E}[U]$ units of time to move from state q to state 0. By [4.3.7], $\mathsf{E}[U] = \mathsf{E}[S]/(1-\rho)$, so we again obtain (5.4.1), but now by proper reasoning.

i.e., not controlled by an N-policy.

The third idea is the most powerful. Write Y for the number of jobs that arrive during a service time. Then, $T(q)$ satisfies the relation

$$T(q) = \mathsf{E}[S] + \mathsf{E}\big[T(q+Y-1)\big], \quad q \geq 1, \tag{5.4.2}$$

because first the job in service must leave, and then, when $Y = k$, it takes $T(q+k-1)$ to clear the system. To solve this equation, we substitute the guess $T(q) = aq + b$ and solve for a and b. It is clear that $T(0) = 0$, hence $b = 0$. Substituting aq into (5.4.2) gives $qa = \mathsf{E}[S] + aq + a\,\mathsf{E}[Y] - a$. Noting that $\mathsf{E}[Y] = \lambda\,\mathsf{E}[S]$, and solving for a we find (5.4.1). We note that the solution of (5.4.2) is unique once $T(0)$ is fixed.

If the system is empty, it takes no time to clear it.

TO FIND THE cost to clear the queue, we first concentrate on the expected queueing cost $U(q)$ that accrue during a service that starts with q jobs in the system. The expected cost for the q jobs already present is $hq\,\mathsf{E}[S]$, i.e., h times the area of the rectangle with base $\mathsf{E}[S]$ and height q. The expected cost for new arrivals during the service is $h\,\mathsf{E}\left[\int_0^S N(s)\,\mathrm{d}s\right]$, where $N(s) \sim \mathrm{Pois}(\lambda s)$. Conditional on $S = t$, $\mathsf{E}\left[\int_0^t N(s)\,\mathrm{d}s\right] = \int_0^t \mathsf{E}[N(s)]\,\mathrm{d}s = \int_0^t \lambda s\,\mathrm{d}s = \lambda t^2/2$, hence, $\mathsf{E}\left[\int_0^S N(s)\,\mathrm{d}s\right] = \mathsf{E}\left[\lambda S^2/2\right]$. The total cost is therefore

Poisson arrivals $\Longrightarrow$ $\mathsf{E}[Y|S=s] = \lambda s \Longrightarrow \mathsf{E}[Y|S] = \lambda S$ $\Longrightarrow$ $\mathsf{E}[Y] = \mathsf{E}[\mathsf{E}[Y|S]] = \lambda\,\mathsf{E}[S]$.

$$U(q) = hq\,\mathsf{E}[S] + \frac{1}{2}\lambda h\,\mathsf{E}\left[S^2\right].$$

Let $V(q)$ be the expected queueing costs to clear the system just after a service starts and q jobs are in the system. By analogy with (5.4.2), $V(q)$ must be the solution of

$$V(q) = U(q) + \mathsf{E}\big[V(q+Y-1)\big], \quad q \geq 1. \tag{5.4.3}$$

Now note that as in (5.4.7), $U(q)$ has a term linear in q and a constant term. We substitute the form $V(q) = aq^2 + bq + c$, assemble terms with the same power in q, and solve for a, b and c. Of course $c = 0$ since $V(0) = 0$. For a and b we need to do some work to arrive at

[5.4.14]–[5.4.15]

$$V(q) = \frac{h}{2}\frac{\mathsf{E}[S]}{1-\rho}q^2 + h\frac{1+\rho C_s^2}{2}\frac{\mathsf{E}[S]}{(1-\rho)^2}q.$$

IT REMAINS TO analyze the long-run average cost under a general N-policy. As we already have expressions for the time and cost while the server is on, we only have to consider the time and cost while the server is off. Clearly, right after the server switches off, we need N independent inter-arrival times to reach level N, which takes N/λ units of time in expectation. As a result, the expected cycle duration is

$$C(N) = N/\lambda + T(N).$$

For the cost during the build up the queue, we use again a recursive procedure. Write $W(q)$ for the accumulated queueing cost from the moment the server becomes idle up to the arrival time of the qth job (the job that sees $q-1$ jobs in the system upon arrival). Then,

Here W is not the waiting time in queue.

[5.4.11]

$$W(q) = W(q-1) + h\frac{q-1}{\lambda} = h\frac{q(q-1)}{2\lambda}.$$

By combining all the above cost factors and using the renewal reward theorem, we find for the long-run average cost

[5.4.16]

$$\frac{W(N)+K+V(N)}{C(N)} = h\frac{1+C_s^2}{2}\frac{\rho^2}{1-\rho} + h\rho + h\frac{N-1}{2} + K\frac{\lambda(1-\rho)}{N}. \tag{5.4.4}$$

This formula provides us with two corner-stone results, one in inventory theory, the other in queueing.

Theorem 5.4.1 (Economic Production Quantity (EPQ) formula)**.** Consider a production-inventory system in which a machine switches on at cost K and items pay holding cost h per unit time. The optimal production quantity is, up to rounding,

$$N^* = \sqrt{\frac{2\lambda(1-\rho)K}{h}}.$$

Proof. In (5.4.4) take the derivative with respect to N, just as if it is a continuous variable. Set the derivative to zero and solve for N. □

Corollary 5.4.2 (Economic Order Quantity (EOQ) formula)**.** If the machine has infinite production speed, the EPQ reduces to $N^* = \sqrt{2\lambda K/h}$, i.e., the EOQ.

Proof. In the EPQ formula, take $\mathsf{E}[S] \to 0$ so that $\rho \to 0$. □

Theorem 5.4.3 (Pollaczek-Khinchine (PK) formula)**.** The expected waiting time of the $M/G/1$ queue is given by

$$\mathsf{E}[W] = \frac{1}{2}\frac{\lambda \mathsf{E}\left[S^2\right]}{1-\rho} = \frac{1+C_s^2}{2}\frac{\rho}{1-\rho}\mathsf{E}[S]. \tag{5.4.5}$$

Proof. In the $M/G/1$ queue, waiting customers pay $h = 1$ per unit time per customer, and the setup cost $K = 0$. Hence, it is optimal to take $N = 1$ in (5.4.4). Next, recall that $\mathsf{E}[L] = \mathsf{E}[Q] + \mathsf{E}[L_s]$, and $\mathsf{E}[Q] = \lambda \mathsf{E}[W]$ and $\mathsf{E}[L_s] = \lambda \mathsf{E}[S]$. Now, realize that the LHS in (5.4.4) is $\mathsf{E}[L]$, while the RHS is $\mathsf{E}[Q] + \mathsf{E}[L_s]$. □

LET US REDERIVE the PK formula by starting from the simple observation that before an arriving job gets access to the server, it first has to wait until the job in service (if any) completes, and then it has wait for the queue to be cleared. From PASTA it follows that $\mathsf{E}[W] = \mathsf{E}[S_r] + \mathsf{E}[Q]\,\mathsf{E}[S]$, where $\mathsf{E}[S_r]$ is the (time-)average remaining service time of the job in service. With Little's law $\mathsf{E}[Q] = \lambda \mathsf{E}[W]$ and writing $\rho = \lambda \mathsf{E}[S]$, we find that

Compare the derivation leading to (5.3.1).

$$\mathsf{E}[W] = \frac{\mathsf{E}[S_r]}{1-\rho}.$$

To make further progress we need to find an expression for $\mathsf{E}[S_r]$ when S is a generally distributed service time. For this, we use the renewal reward theorem, just as in Section 5.3.

For the $M/M/1$ queue, $\mathsf{E}[S_r] = \rho \mathsf{E}[S]$, but not for the $M/G/1$ queue.

Theorem 5.4.4 (Remaining Service time)**.** The expected remaining service time as observed by an arrival is

$$\mathsf{E}[S_r] = \frac{\lambda}{2}\mathsf{E}\left[S^2\right],$$

provided the second moment of the (generic) service time S exists.

Proof. Consider the kth job of some sample path of the $M/G/1$ queueing process. Let the job's service start at time $\tilde{A}_k$, so that it departs at time $D_k = \tilde{A}_k + S_k$, see Fig. 5.4.1. At time s, the remaining service time of job k is $(D_k - s)\,\mathbb{1}_{\tilde{A}_k \le s < D_k}$. More generally, at time s, the number of departures is $D(s)$. Thus, $D(s)+1$ is the index of the first job to depart after time s, and its departure time is $D_{D(s)+1}$. Consequently, the remaining service time at time s is $D_{D(s)+1} - s$, provided this job is in service. All in all, the total remaining service time as seen by the server up to time t is given by

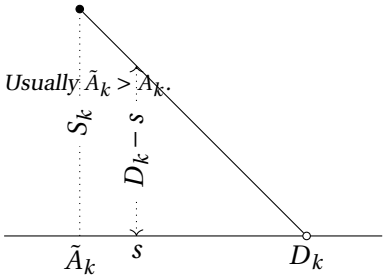


Fig. 5.4.1: Remaining service time.

$$Y(t) = \int_0^t (D_{D(s)+1} - s)\,\mathbb{1}_{A_{D(s)+1} \le s \le D_{D(s)+1}}\,\mathrm{d}s$$

As $t \to \infty$, $Y(t)/t \to \mathsf{E}[S_r]$, i.e., the (time-average) remaining service time.

We also see in Fig. 5.4.1 that $Y(D_k) - Y(D_{k-1}) =: X_k$ is the area under the triangle. By choosing $T_k = D_k$ in the renewal-reward theorem as the epochs to inspect $Y(\cdot)$, $X = \lim_{n\to\infty} \frac{1}{n}\sum_{k=1}^{n} S_k^2/2 = \mathsf{E}\left[S^2\right]/2$. By the renewal-reward theorem, $Y = \delta X$, but $\delta = \lambda$; the result follows directly. □

By rate-stability.

Replacing this expression in the above expression for $\mathsf{E}[W]$, we obtain the PK formula from observing that

$$\frac{\mathsf{E}\left[S^2\right]}{(\mathsf{E}[S])^2} = \frac{\mathsf{E}\left[S^2\right] - (\mathsf{E}[S])^2 + (\mathsf{E}[S])^2}{(\mathsf{E}[S])^2} = \frac{\mathsf{V}[S] + (\mathsf{E}[S])^2}{(\mathsf{E}[S])^2} = C_s^2 + 1.$$

THE POLLACZEK-KHINCHINE EQUATION can also be found as a limiting case of the expression (5.3.3) of the waiting time of the $M^B/M/1$ queue. Recall that, in the $M^B/M/1$ queue, batches of items arrives, and the service times of the items are iid with mean $1/\nu$, say. If F is the cdf of the service times of the $M/G/1$ queue, then we take

If you are not interested in maths, than you should this derivation.

$$\mathsf{P}\{B = k\} = F(k/\nu) - F((k-1)/\nu)$$

as the probability that a batch has size k. Note that when a batch has size k, the expected service time is the sum of k rvs that are $\text{Exp}(\nu)$; thus, the service time of a batch of size k is $\sim \text{Gamma}(k,\nu)$. With this, the expected service time of a batch becomes

$$\begin{aligned}\mathsf{E}[S_B] &= \mathsf{E}[\mathsf{E}[S_B|B]] = \mathsf{E}[B/\nu] = \sum_{1=0}^{\infty} k/\nu\, \mathsf{P}\{B = k\} \\ &= \sum_{1=0}^{\infty} k/\nu [F(k/\nu) - F((k-1)/\nu)] \approx \int_0^{\infty} x\, \mathrm{d}F(x) = \mathsf{E}[S]\,.\end{aligned}$$

In words, when ν becomes large, the expected service time of a batch in the $M^B/M/1$ queue is nearly the the same as the expected waiting time of a job of the $M/G/1$ queue.

In fact, the proof that the expected waiting time of the $M^B/M/1$ converges to the Pollaczek-Khinchine equation when $\nu \to \infty$ is based on the fact that every cdf F, concentrated on $[0,\infty)$, can be approximated (weakly) by the sequence of cdfs

See Asmussen, Section III.4, Applied Probability and Queues, 2003.

$$F_\nu(x) = F(0) + \sum_{k=1}^{\infty} [F(k/\nu] - F((k-1)/\nu)] \frac{e^{-\nu t}}{x} \frac{(\nu x)^k}{(k-1)!}, \quad \text{as } \nu \to \infty.$$

That is to say, the distribution of the service times of the batches whose item times are $\text{Exp}(\nu)$ approaches the cdf of the service times of the $M/G/1$ queue. This implies that the expectation and scv off the batch service times converges to $\mathsf{E}[S]$ and C_S^2 of F. Finally, the last term in (5.4.5) becomes small because the service time of a single item becomes negligible as $\nu \to \infty$.

TF 5.4.1. Consider the $M/G/1$ queue. Denote by $\tilde{A}_k$ the time job k starts service and by D_k its departure time, $k = 1,\dots,n$. Claim: the expression

$$\sum_{k=1}^{n} \int_0^{D_n} (D_k - s)\, \mathbb{1}_{\tilde{A}_k \le s < D_k}\, \mathrm{d}s$$

computes the total remaining service time up to time $t = D_n$.

TF 5.4.2. Claim: For an $M/D/1$ queue the expected waiting time is $\mathsf{E}[W] = \frac{\rho}{1-\rho}\frac{\mathsf{E}[S]}{2}$.

TF 5.4.3 (5.6)**.** A machine can switch on and off. If the queue length hits N, the machine switches on, and if the system becomes empty, the machine switches off. Let $I_k = 1$ if the machine is on in period k and $I_k = 0$ if it is off, let L_k be the number of items in the system at the end of period k. Claim: the next recursions model this queueing system.

$$I_{k+1} = \begin{cases} 1 & \text{if } L_k \geq N, \\ I_k & \text{if } 0 < L_k < N, \\ 0 & \text{if } L_k = 0, \end{cases}$$

$$\begin{aligned} I_{k+1} &= \mathbb{1}_{L_k \geq N} + I_k \mathbb{1}_{0<L_k<N}, \\ d_k &= \min\{L_{k-1}, c_k\}, \\ L_k &= L_{k-1} - (1 - I_k) d_k + a_k. \end{aligned}$$

Assume that $I_0 = 0$ at time $k = 0$.

TF 5.4.4 (5.6)**.** We have an $M/G/1$ queue controlled by an N-policy, $\lambda = 1$ and $S_i \equiv 2$. Claim, the expected time to clear the system after reaching $N-$jobs is the same as for as a $D/D/1$ queue with the same arrival and service rate.

Ex 5.4.5. Explain intuitively that the system is rate-stable for any N.

Ex 5.4.6. Why doesn't the utilization ρ depend on N?

Ex 5.4.7. Consider a workstation with just one machine. We model the job arrival process as a Poisson process with rate $\lambda = 3$ per day. The average service time $\mathsf{E}[S] = 2$ hours, $C_s^2 = 1/2$, and the shop is open for 8 hours per day. Show that $\mathsf{E}[W] = 4.5h$. What would you propose to reduce $\mathsf{E}[W]$ to 2h?

Ex 5.4.8. Compare the expected waiting time of the $M/D/1$ queue to that of the $M/M/1$ queue.

Ex 5.4.9. Compute $\mathsf{E}[J]$ for the $M/G/1$ queue with $S \sim U[0, \alpha]$.

Ex 5.4.10. Show for the $M/G/1$ that $\mathsf{E}[S_r] = \rho\,\mathsf{E}[S_r \mid S_r > 0]$. As consequence, for the $M/M/1$ queue, $\mathsf{E}[S_r] \neq \mathsf{E}[S]$; why not?

Ex 5.4.11. Explain the recursion for $W(q)$ and solve it.

Ex 5.4.12. Use the memoryless property for the $M/M/1$ queue to show that

$$T(q) = \frac{1}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} T(q+1) + \frac{\mu}{\lambda+\mu} T(q-1). \tag{5.4.6}$$

Ex 5.4.13. Show that for the $M/M/1$ queue, $V(q)$ satisfies a relation like this:

$$V(q) = h\frac{q}{\lambda+\mu} + \frac{\lambda}{\lambda+\mu} V(q+1) + \frac{\mu}{\lambda+\mu} V(q-1). \tag{5.4.7}$$

Ex 5.4.14. Simplify $aq^2 + bq = a\mathsf{E}\left[(q+Y-1)^2\right] + b\mathsf{E}\left[q+Y-1\right] + U(q)$, where $U(q) = hq\,\mathsf{E}[S] + \frac{1}{2}\lambda h\,\mathsf{E}\left[S^2\right]$, by assembling powers in q to obtain:

$$\begin{aligned} a &= \frac{h}{2}\frac{\mathsf{E}[S]}{1-\mathsf{E}[Y]} = \frac{h}{2}\frac{\mathsf{E}[S]}{1-\rho}, \\ b(1-\mathsf{E}[Y]) &= a(\mathsf{E}\left[Y^2\right] - 2\mathsf{E}[Y] + 1) + \frac{1}{2}h\lambda\,\mathsf{E}\left[S^2\right]. \end{aligned}$$

Th 5.4.15. Derive the expression for $V(q)$ with the previous exercises.

Th 5.4.16. Derive (5.4.4).

5.5 INVENTORY CONTROL, ANALYTIC RESULTS

In this section we establish a set of formulas by which we can compute several performance measures for the single-item inventory systems discussed in Section 2.3. With these formulas we can avoid simulation to obtain insight into the performance of the basestock and the (Q,r) inventory control policies as function of the policy parameters. However, we will not deal with the (s,S) policy because this is a bit too difficult for these notes.

For those interested in a proof of the optimality of (s,S) policies and an algorithm to compute the optimal policy parameters see Van Foreest and Kilic [2023].

We need some practical notation. The period demands $\{D_i\}_{i=1}^{\infty}$ are assumed to be iid rvs; for notational ease we take $D_i = 0$ for $i \le 0$. Write

$$D(i,j] = \sum_{k=i+1}^{j} D_k, \qquad D[i,j] = D_i + D(i,j], \qquad D[i,j) = D[i,j] - D_j,$$

and likewise for the amount of out-standing reorders $Q(i,j]$.

We assume that the leadtime L is constant and a multiple of the period duration. Since the period demand are iid rvs, the rvs $\{D[k-L,k)\}_{k=L+1}^{\infty}$ have the same distribution. Let us write $F(x)$ for the common cdf $\mathsf{P}\{\sum_{i=1}^{L} D_i \le x\}$, and let $X \sim F$. Observe that $\mathsf{E}[X] = L\mathsf{E}[D]$. From F we can easily obtain the pmf f and the survivor function $G = 1 - F$.

Mind, they are not necessarily independent.

FOR THE BASESTOCK model in discrete time, recall from (2.3.2) that the start-of-period inventory position P_k is always equal to the order-up-to level S, the end-of-period inventory position $P'_k = S - D_k$, and the inventory level, $I_k = I'_{k-1} + Q_{k-L}$. We are going to use these recursions to find expressions for the performance measures when the inventory system has been running for at least one lead time L.

To get rid of the initial effects.

The first step is to relate the outstanding orders to the demand.

Lemma 5.5.1. Under a basestock policy with order-up-to-level S, if there are no outstanding orders at period $k = 1$, then for all periods $k \ge 1$,

$$Q(1,k] = D[1,k).$$

That is, for some period $k > L$, the demand $D[1,k-L) = Q(1,k-L]$ has been delivered, and the rest $D[k-L,k) = Q(k-L,k]$ is still under way.

Proof. Using the recursions for the inventory position under the basestock policy repeatedly,

$$\begin{aligned} P_k &= P'_{k-1} + Q_k = P_{k-1} - D_{k-1} + Q_k \\ &= P_{k-2} - D_{k-2} - D_{k-1} + Q_{k-1} + Q_k \\ &= P_1 - D[1,k) + Q(1,k]. \end{aligned}$$

The proof is finished by realizing that $P_k = S$ for all $t \ge 1$, and using that $D_i = 0$ for $i \le 0$ and there are no outstanding orders from periods before $k = 1$. □

We can use this lemma to express I_k in terms of S and the demand during the lead time $D[k-L,k)$.

Lemma 5.5.2. Under a basestock policy, if the inventory starts at $I_1 = S$ and there are no outstanding orders of period $t \le 1$, then for all periods $t \ge 1$,

$$I_k = S - D[k-L,k), \qquad I'_k = I_k - D_k = S - D[k-L,k].$$

In words, items not on-hand must be on order.

Proof. Using the recursion for the inventory level and the assumptions,

$$\begin{aligned} I_k &= I'_{k-1} + Q_{k-L} = I_{k-1} - D_{k-1} + Q_{k-L} \\ &= I_{k-2} - D_{k-2} - D_{k-1} + Q_{k-L-1} + Q_{k-L} \\ &= I_1 - D[1,k) + Q(1,k-L] \\ &\overset{1}{=} I_1 - D[1,k) + D[1,k-L) \\ &\overset{2}{=} S - D[k-L,k); \end{aligned}$$

1 uses Lem. 5.5.1, 2 the assumption $I_1 = S$. □

This next theorem, which allows us to properly define the inventory level as an rv I, follows directly from the above lemmas.

Theorem 5.5.3. The rvs $\{I_k\}_{k=L+1}^{\infty}$ are identically distributed as the rv

But not necessary independent

$$I = S - X,$$

with $X = \sum_{i=1}^{L} D_i$.

Proof. As observed earlier, all rvs $D[k-L,k)$ have the same cdf with common rv X. As S is constant, the rvs $I_k = S - D[k-L,k) \sim S - X$ for all $k \ge L+1$. The same reasoning applies to I'_k. □

Now that we have characterized the distribution I in terms of a sum of L iid demands, we can find expressions for the performance measures. First, for the ready rate,

$$\alpha \overset{1}{=} \mathsf{P}\{I' \ge 0\} \overset{2}{=} \mathsf{P}\{I - D \ge 0\} \overset{3}{=} \mathsf{P}\{S - X - D \ge 0\} = \mathsf{P}\{X + D \le S\}, \quad (5.5.1)$$

where 1 follows from the definition of α, 2 from $I' = I - D$, 3 from $I = S - X$. Second, by taking expectations in (2.3.3), we obtain for the fill rate

$I^+ := [I]^+ := \max\{I, 0\}$,
$I^- := [-I]^+$.

$$\beta = \frac{\mathsf{E}\left[\min\{D, I^+\}\right]}{\mathsf{E}[D]}. \quad (5.5.2)$$

For the third performance measure, the cycle service level, I have not yet been able to find a simple expression in terms of expectations.

I also did not push.

The mean inventory level is simple,

$$\mathsf{E}[I] = \mathsf{E}[S - X] = S - \mathsf{E}[X] = S - L\mathsf{E}[D].$$

Next, noting that $I^+ - I^- = I$, the average on-hand inventory can be expressed in terms of the average backlog:

$$\mathsf{E}\left[I^+\right] = \mathsf{E}[I] + \mathsf{E}[I^-] = S - L\mathsf{E}[D] + \mathsf{E}[I^-].$$

From the definition of the backlog,

$$\mathsf{E}[I^-] = \mathsf{E}\left[[-I]^+\right] = \mathsf{E}\left[[X-S]^+\right] = \sum_{k=0}^{\infty} (k-S)^+ f(k). \tag{5.5.3}$$

Clearly, this summation runs to ∞ if the support of D is unbounded. As this unpractical for numerical purposes, let's rewrite it to a more manageable form.

Lemma 5.5.4. The average backlog is equal to

$$\mathsf{E}[I^-] = L\,\mathsf{E}[D] - \sum_{j=0}^{S-1} G(j),$$

where $G(\cdot)$ is de sf of X.

Proof. Since $\sum_{j=0}^{\infty} \mathbb{1}_{j<k} = k$, we find from (5.5.3)

$$\begin{aligned}\sum_{k=S}^{\infty} (k-S) f(k) &= \sum_{k=S}^{\infty}\sum_{j=0}^{\infty} \mathbb{1}_{j<k-S} f(k) = \sum_{j=0}^{\infty}\sum_{k=S}^{\infty} \mathbb{1}_{k>j+S} f(k) \\ &= \sum_{j=0}^{\infty} G(j+S) = \sum_{j=0}^{\infty} G(j) - \sum_{j=0}^{S-1} G(j).\end{aligned}$$

Because $\sum_{k=0}^{\infty} G(i) = L\,\mathsf{E}[D]$ the claim follows. □

Recall from Th. 5.5.3 that I, hence I^+ and I^-, are functions of the reorder level S. Therefore, the average inventory cost (on-hand and backlogging combined)

$$c(S) = h\,\mathsf{E}\left[I^+\right] + b\,\mathsf{E}[I^-],$$

is also a function of S.

Theorem 5.5.5. The cost function $S \to c(S)$ is convex and coercive. Thus, there exists an optimal order-up-to level S^* that minimizes the average cost.

Make a drawing to see this.

Proof. For fixed i, the function $S \to [S-i]^+$ is convex. Since a weighted sum of convex functions is still convex, $\sum_{i=0}^{\infty} f(i)[S-i]^+$ is also convex. But $\sum_{i=0}^{\infty} f(i)[S-i]^+ = \mathsf{E}\left[[S-I]^+\right] = \mathsf{E}\left[I^+\right]$, and therefore $h\,\mathsf{E}\left[I^+\right]$ is a convex function of S. Similarly, $S \to [i-S]^+$ is convex, and therefore $\mathsf{E}[I^-]$ is convex. Consequently, $c(S)$, being the sum of two convex functions, is itself convex. Finally, $\mathsf{E}\left[I^+\right] \to \infty$ when $S \to \infty$, and $\mathsf{E}[I^-] \to \infty$ when $S \to -\infty$; therefore $c(\cdot)$ has a minimum. □

Here is an illustration how to code the Lighthouse Company example of Section 2.3. With the random variable class we built in Section 1.4, our code can stay nearly in one-to-one correspondence with the mathematical definitions above. Note that the performance measures depend on the control parameter S, i.e., the order-up to level).

```
from functools import cache
import numpy as np

import random_variable as rv
```

The basestock policy follows straightaway once we realize that we just have to specify the rv X, and that we can derive the other rvs from X.

```
class Basestock:
    def __init__(self, D, L, h, b):
        self.D = D
        self.L = L
        self.h = h
        self.b = b
        self.X = sum(self.D for i in range(self.L))

    @cache
    def IP(self, S):
        return S

    @cache
    def IL(self, S):
        return self.IP(S) - self.X

    @cache
    def Imin(self, S):
        return rv.apply_function(lambda x: max(0, -x), self.IL(S))

    @cache
    def Iplus(self, S):
        return rv.apply_function(lambda x: max(0, x), self.IL(S))

    def cost(self, S):
        return self.b * self.Imin(S).mean() + self.h * self.Iplus(S).mean()

    def alpha(self, S):
        return (self.IL(S) - self.D).sf(-1)

    def beta(self, S):
        m = rv.compose_function(lambda x, y: min(x, y), self.D, self.Iplus(S))
        return m.mean() / self.D.mean()
```

We instantiate a basestock object like this.

```
# Daily demand
D = rv.RV({1: 1 / 6, 2: 1 / 5, 3: 1 / 4, 4: 1 / 8, 5: 11 / 120, 6: 1 / 6})
L = 2
c = 100  # buying price
b = 0.1 * c  # backlog
h = 0.5 * c / 30  # holding
S = 5

base = Basestock(D, L, h, b)
```

Here are some tests, which as a matter of principle, we should not skip. Realize that this serves not only as a test of the implementation, but also of the above algebra.

```
theta = L * D.mean()
assert np.isclose(base.IL(S).mean(), S - theta)
assert np.isclose(
```

```
    base.Imin(S).mean(), theta - sum(base.X.sf(j) for j in range(0, S))
)
assert np.isclose(base.alpha(S), (base.X + D).cdf(S))
```

To run it, use the next example code.

```
for S in range(-2, 8):
    print(base.Imin(S).mean(), base.Iplus(S).mean(), base.cost(S))
    print(base.alpha(S), base.beta(S))
```

THE (Q,r) MODEL is our next target. There is a nice way to relate the (Q,r) system to a number of basestock systems. To see this, imagine two animals, a squirrel that observes the (Q,r) system at the end of each period, and a bear that only wakes up when the inventory position right at the start of the interval equals some fixed i, where i must lie in the set $\{r+1,\dots r+Q\}$; if the position is not equal to k the bear hibernates. Clearly, from the bear's point of view, the system behaves as a basestock system with order-up-to level i, because each period the bear is awake, the inventory position is i, and the position just before the bear falls asleep is $i-D$. More generally, we can associate a different bear to each position in the set $\{r+1,\dots r+Q\}$, so that always exactly one bear is awake, and the other bears sleep. To combine the statistics as observed by the bears and the statistics as observed by the squirrel, we need to know for each i the average fraction of periods the inventory position P equals i. For instance, if the squirrel knows

We assume that the limit over all sample paths $\{P_i\}$ exists almost surely, so that we can invoke the strong law of large numbers.

$$p(i) = \lim_{k\to\infty} \frac{1}{T}\sum_{k=1}^{T} \mathbb{1}_{P_k=i},$$

then it can retrieve the ready rate $\sum_{i=r+1}^{Q+r} p(i)\alpha(i)$ from the ready rate $\alpha(i)$ as seen by bear with order-up-to level i.

We now need to label the ready rate of the basestock model by the order-up-to level.

Theorem 5.5.6. Under the (Q,r) policy the inventory position $P \sim \text{Unif}r+1,\dots,r+Q$ in stationarity, that is, $p(i) = \mathsf{P}\{P=i\} = 1/Q$ for $i = r+1,\dots,r+Q$. Consequently, I is distributed as $P-X$.

Proof. For ease take $Q=3, r=0$, so that the inventory position P cycles between the levels $1,2$ and 3. We can use level crossing arguments as follows. Write $f_i = \mathsf{P}\{D \in \{i, 3+i, 6+i, \dots\}\}$ for the probability that the period demand is a multiple of 3 offset by i, $i=0,1,2$. Suppose that, at the start of period k, the position $P_k = 3$. Since we order in multiples of $Q=3$, the probability $\mathsf{P}\{P_{k+1}=3|P_k=3\} = f_0$, i.e., equal to the probability that D_k is a multiple of 3. Likewise, $\mathsf{P}\{P_{k+1}=2|P_k=3\} = f_1$, and $\mathsf{P}\{P_{k+1}=1|P_k=3\} = f_2$. However, again because we order in multiples of Q, $\mathsf{P}\{P_{k+1}=1|P_k=2\} = f_1$, and similarly for the other possibilities. Therefore, we can use level-crossing to conclude that on the long run, $(f_1+f_2)p(i) = f_1 p(2) + f_2 p(1)$. But, because of the (Q,r) ordering rule, we can cyclically change $p(1), p(2)$ and $p(3)$ in this equality. The only solution that is compatible with symmetry is that $p(i) = 1/3$ for all i. □

The proof is the same for general Q and r, but needs a bit more notation.

With the above theorem, the ready rate for the (Q,r) model can be computed with the formula

$$\alpha = \frac{1}{Q}\sum_{i=r+1}^{r+Q}\alpha(i) = \frac{1}{Q}\sum_{i=r+1}^{r+Q}\mathsf{P}\{X+D\le i\}. \tag{5.5.4}$$

If we have the distribution of I, which we can obtain numerically with our random variable class, see below, we can compute the fill rate in accordance with (5.5.2).

For the average inventory level, we write $I(i)$ for the the inventory level of the basestock model with order-up-level i, and get

$$\mathsf{E}[I] = \frac{1}{Q}\sum_{i=r+1}^{r+Q}\mathsf{E}[I(i)] = \frac{1}{Q}\sum_{i=r+1}^{r+Q}(i-\mathsf{E}[X]) = \frac{Q+1}{2}+r-\mathsf{E}[X],$$

The expected number of back-orders $\mathsf{E}[I^-]$ and expected on-hand inventory $\mathsf{E}\left[I^+\right]$ can be found be equivalent summations. Finally, with h and b the holding and backorder cost per item per period, the total average cost becomes

$$c(Q,r) = K\frac{\mathsf{E}[D]}{Q} + h\mathsf{E}\left[I^+\right] + b\mathsf{E}[I^-].$$

With regard to the first term, note that $\mathsf{E}[D]$ is average demand per period, so $\mathsf{E}[D]/Q$ is the order frequency.

It remains to find good values for r and Q. There are some fast algorithms available to compute optimal r and Q, but we don't discuss these here. A simple work around is to carry out a full grid search over a (reasonable) set of pairs of r and Q. Another simple heuristic is to take $Q = \sqrt{2\mathsf{E}[D]K/h}$, i.e., the EOQ value, and then search for r such that the total average cost is minimal. With similar methods we can search for an r such that the α or β service level criteria are met.

To code the above, we use the random variable class again so that our implementation can stay on a high level. Note how the code uses Th. 5.5.6. If you compare the implementation of the basestock and the (Q,r) classes, you'll see how much they resemble each other.

```
78  class Qr:
79      def __init__(self, D, L, h, b):
80          self.D = D
81          self.L = L
82          self.h = h
83          self.b = b
84          self.X = sum(self.D for i in range(self.L))
85
86      @cache
87      def IP(self, Q, r):
88          return rv.RV({i: 1 / Q for i in range(r + 1, r + Q + 1)})
89
90      @cache
91      def IL(self, Q, r):
92          return self.IP(Q, r) - self.X
93
94      @cache
95      def Imin(self, Q, r):
```

```
96          return rv.apply_function(lambda x: max(0, -x), self.IL(Q, r))
97
98      @cache
99      def Iplus(self, Q, r):
100         return rv.apply_function(lambda x: max(0, x), self.IL(Q, r))
101
102     def cost(self, Q, r):
103         return (
104             self.b * self.Imin(Q, r).mean() + self.h * self.Iplus(Q, r).mean()
105         )
106
107     def alpha(self, Q, r):
108         return (self.IL(Q, r) - self.D).sf(-1)
109
110     def beta(self, Q, r):
111         m = rv.compose_function(
112             lambda x, y: min(x, y), self.D, self.Iplus(Q, r)
113         )
114         return m.mean() / self.D.mean()
```

To run it, use the next example code.

```
121 qr = Qr(D, L, h, b)
122 Q, r = 3, 10
123 print(qr.IL(Q, r).mean())
124 print((Q + 1) / 2 + r - qr.X.mean())
```

TF 5.5.1. Customers of fast-food restaurants prefer to be served from stock. For this reason such restaurants often use a 'produce-up-to' policy with level S: When the on-hand inventory I is equal or lower than $S-1$, the company produces items until the inventory level equals S again.

Suppose that customers arrive as a Poisson process with rate λ and the production times of single items are iid and exponentially distributed with parameter $\mu > \lambda$. Assume also that customers who cannot be served from on-hand stock are backlogged, that is, they wait until their item has been produced.

Claim: The average on-hand inventory level is S minus the average number of hamburges the cook is baking. i.e., $\mathsf{E}[I] = \sum_{i=0}^{S}(S-i)\,p(i)$, where $p(i) = (1-\rho)\rho^i$ with $\rho = \lambda/\mu$.

Ex 5.5.2. TODO Here is a subtle problem, that you should think about, and *memorize*. In these notes so far, we have been concerned with *periodic-review* systems: at the end of period k, we check the inventory I'_k. Hence, if $I'_k \geq 0$, all demand must have been satisfied.

In continuous-review systems, the notation I_k corresponds to the inventory level as perceived by the kth customer (observe that this is very different from the meaning of the I_k for periodic-review systems).

Under a basestock policy, why is the ready rate $F(S)$ (and not $F(S) = F(s+1)$ as in (4.5.3)) for continuous-review systems?

6

APPROXIMATE MODELS

In the previous two chapters we learned how to construct and simulate queueing processes. Simulation is a powerful tool but one of its limitations is that it does not easily provide insight into structural behavior of systems. For this we need theoretical models, and the derivation of such models form the contents of the remainder of the book.

IN THIS CHAPTER we discuss two formulas that might be considered as the most important formulas to understand the behavior of queueing systems. The first is Sakasegawa's formula that approximates the expected queueing time in a $G/G/c$ queue; the second characterizes the propagation of variability through a tandem network of $G/G/c$ queues. With a bit of exaggeration, we can say that the entire philosophy behind lean manufacturing and the world-famous Toyota production system are based on the principles that can be derived from these two formulas.

Here we take these formulas for granted, but focus on the insights they provide into the performance of queueing systems and how to use them to guide improvement procedures for production and service systems.

In Section 6.1 we introduce Sakasegawa's formula and discuss the main insights it offers. Then we illustrate how to use this formula to estimate waiting times in three queueing settings in which the service process is interrupted. In the first case, Section 6.2, the server has to produce jobs from different families, and there is a change-over time required to switch from one production family to another. As such setups reduce the time the server is available, the load must increase. In fact, to reduce the load, the server produces in batches of fixed sizes. In the second case, in Section 6.3, the server sometimes requires small adjustments, for instance, to prevent the production quality to degrade below a certain level. Clearly, such adjustments are typically not required during a job's service; however, they can occur *between* any two jobs. As a consequence, the number of jobs served between two such adjustments (or setups) is not constant, hence different from batch production where batch sizes are constant. In the third example, in Section 6.4, quality problems or break downs can occur *during* a job's service. These make job service times more variable, which leads to longer expected queueing times. In the final Section 6.5, we concentrate on tandem queues.

In passing, we use some interesting results of probability theory and the Poisson process.

6.1 $G/G/c$ QUEUE: SAKASEGAWA'S FORMULA

While there is no expression available to compute the exact expected waiting time for the $G/G/c$ queue, Sakasegawa's formula provides a reasonable approximation. This takes the form

$$\mathsf{E}[W] = \frac{C_a^2 + C_s^2}{2} \frac{\rho^{\sqrt{2(c+1)}-1}}{1-\rho} \frac{\mathsf{E}[S]}{c}, \tag{6.1.1}$$

where $C_a^2 = \mathsf{V}[X]/(\mathsf{E}[X])^2$ and $C_s^2 = \mathsf{V}[S]/(\mathsf{E}[S])^2$ are the square coefficient of variation (SCV) of the inter-arrival time and service time, respectively, and the utilization of the station is $\rho = \lambda \mathsf{E}[S]/c$, where λ is the rate at which jobs arrive at the system, $\mathsf{E}[S]$ is the expected service time, and c the number of servers. It is evident from this formula that $\rho \in [0,1)$, hence the load and utilization are the same.

Here is a subtle point. When there are multiple machines, the utilization of each machine need not be equal to ρ. For instance, if there is a preference to choose the 'left-most' machine whenever it is free, then the utilization of this machine is larger than ρ. Only when the machines have the same speed, and the routing *is such that each machine receives a fraction λ/c of jobs, the machines have the same utilization.*

Note that Sakasegawa's formula for the $G/G/c$ queue reduces to the PK formula for $\mathsf{E}[W]$ for the $M/G/1$ queue; just fill in $c = 1$ and set $C_a^2 = 1$. Part of the intuition behind Sakasegawa's formula is based on the following idea. To generalize the formula for $\mathsf{E}[W]$ from the $M/M/1$ queue to that of the $M/G/1$ queue, we replace $(1+1)/2$ by $(1+C_s^2)/2$; the result remains exact. Now, to go from the $M/G/1$ queue to the $G/G/1$ queue, replace $(1+C_s^2)/2$ by $(C_a^2+C_s^2)/2$; this is no longer exact, but the approximation remains good for most practical purposes. For the details behind the power with c, we refer to Sakasegawa's paper.

IT IS CRUCIAL to memorize the insights into the performance of queueing systems that this formula offers. Even though (6.1.1) is an approximation, it proves to be exceedingly useful when designing queueing systems and analyzing the effect of certain changes.

First, we see that $\mathsf{E}[W] \sim (1-\rho)^{-1}$. Consequently, when ρ is large, the waiting time is (very) large. And, not only is $\mathsf{E}[W]$ large, it is also extremely *sensitive* to the actual value of ρ. Clearly, such situations must be avoided, and therefore, when trying to improve a queueing system, the first focus should be on reducing ρ.

Second, $\rho = \lambda \mathsf{E}[S]/c$. Thus, when ρ must be made smaller, we have just three options. We can reduce the arrival rate λ of jobs, for instance by blocking demand, or sending it elsewhere such as to another machine. We can make $\mathsf{E}[S]$ smaller by replacing a server with a faster one or by shifting some of the processing steps of a job to other servers, thereby making job sizes smaller. Finally, we can add servers, i.e., making c larger. If technically possible, adapting c to λ when $\mathsf{E}[S]$ cannot be easily changed is a very effective mechanism to control waiting times.

And not more!

And if customers have a choice, they will take their own measures, simply by going elsewhere.

This is precisely what you see in supermarkets.

Third, $\mathsf{E}[W] \sim C_a^2$ and $\mathsf{E}[W] \sim C_s^2$, which implies that when job inter-arrival or service times are very variable, $\mathsf{E}[W]$ is large. Thus, after ensuring that ρ is sufficiently small, it becomes important to concentrate on reducing on C_a^2 and C_s^2.

It works the other way too: systems with low variability can deal with higher load.

Finally, $\mathsf{E}[W] \sim \mathsf{E}[S]/c$. This says that, from the perspective of a job in queue, average job service times are c times as short as 'its own service time'.

TF 6.1.1. Sakasegawa's approximation for the waiting time in a $G/G/c$ queue is

$$\mathsf{E}[W] = \frac{C_a^2 + C_s^2}{2} \frac{\rho^{\sqrt{2(c+1)}-1}}{c(1-\rho)} \mathsf{E}[S].$$

We claim that it is exact for the $M/G/1$ queue.

TF 6.1.2. Claim: For a $G/G/1$ queue, if $\rho = 0.95$ then a 5% increase in the service rate reduces the average waiting time by about a half.

TF 6.1.3. Sakasegawa's approximation for the waiting time in a $G/G/c$ queue is

$$\mathsf{E}[W] = \frac{C_a^2 + C_s^2}{2} \frac{\rho^{\sqrt{2(c+1)}-1}}{c(1-\rho)} \mathsf{E}[S].$$

Claim: For a stable $D/D/1$ queue $C_a^2 = 0$ and $C_s^2 = 0$, therefore $\mathsf{E}[W] = 0$.

Ex 6.1.4. In a manufacturing setting, the Poisson process is not always a suitable model for the arrival process of jobs at a production station. Can you provide an example to see why this is the case?

Ex 6.1.5. Consider a single-server queue at which every minute a customer arrives, precisely at the first second and $S \equiv 50$ s. What are ρ, $\mathsf{E}[L]$, C_a^2, and C_s^2?

This is an important example to memorize.

Ex 6.1.6. Consider the same single-server system as in [6.1.5], but now the customer service time is stochastic: with probability $1/2$ a customer requires 1 minute and 20 seconds of service, and with probability $1/2$ the customer requires only 20 seconds of service. What are ρ, C_a^2, and C_s^2?

Ex 6.1.7. In a workplace, when a mechanic has used a specific high precision tool, the tool has to be inspected by a microscope for potential failures. The inter-arrival times of the tools at the microscope is $X \sim \text{Unif}[3,9]$ minutes, and the inspection time S is such that $\mathsf{E}[S] = 5$ minutes and $\mathsf{V}[S] = 4$ minutes squared. Use Sakasegawa's formula to estimate $\mathsf{E}[W]$. Next, when tools would arrive as a Poisson process, what would be $\mathsf{E}[W]$? Discuss your findings.

Ex 6.1.8. A machine serves two types of jobs. The processing time of jobs of type i, $i = 1,2$, is exponentially distributed with parameter μ_i. The type T of a job is random and independent of anything else, and such that $\mathsf{P}\{T = 1\} = p = 1 - q = 1 - \mathsf{P}\{T = 2\}$. (An example is a desk serving adults and children, both requiring different average service times, and p is the probability that the customer in service is a adult.) Show that the expected processing time and variance are given by

$$\begin{aligned}\mathsf{E}[S] &= p\,\mathsf{E}[S_1] + q\,\mathsf{E}[S_2] \\ \mathsf{V}[S] &= p\,\mathsf{V}[S_1] + q\,\mathsf{V}[S_2] + pq(\mathsf{E}[S_1] - \mathsf{E}[S_2])^2.\end{aligned}$$

Thus, even if $\mathsf{V}[S_1] = \mathsf{V}[S_2] = 0$, still $\mathsf{V}[S] > 0$ if $\mathsf{E}[S_1] \neq \mathsf{E}[S_2]$. Mixing different jobs types increases variability, hence queueing times.

6.2 SERVER SETUPS

In some cases, machines have to be setup before they can start producing items. Consider, for instance, a machine that paints red and blue items. When the machine requires a color change, it may be necessary to clean up the machine, which takes time. Another example is an oven that needs a temperature change when different item types require different production temperatures. Service operations form another setting with setup times: when servers (personnel) have to move from one part of a building to another, the time spent moving cannot be spent on serving customers.

In realistic problems there can be tens of families.

In all such cases, the setups consume a significant amount of time; in fact, setup times of an hour or longer are not uncommon. To prevent overloading the server, it is necessary to produce in batches such that a server first processes a batch of jobs of one type, then performes a setup to serve a batch of another type, and so on.

Often, the setup time depends on the sequence in which the families are produced, for example, a switch in color from white to black might take less cleaning time than from black to white. The problem then becomes to determine a production sequence that minimizes the sum of the setup times to produce the families, and then find suitable batch sizes to minimize the average waiting times.

Here we use Sakasegawa's formula to model the effect of change-over, or setup, times on the average sojourn time of jobs. In the main text we provide a list of elements required to compute $\mathsf{E}[J]$, in [6.2.3] we illustrate in detail how to carry out the computations.

Producing one type of job is often called a 'run'.

ASSUME A SINGLE machine produces runs of red jobs and blue jobs, and needs a setup for each color change. Jobs arrive at rate λ_r and λ_b. The scv of both job types inter-arrival times is given by C_a^2. Once the server is setup for the correct color, we assume for simplicity that jobs of both type have the same average *net processing time* $\mathsf{E}[S_0]$ and variance $\mathsf{V}[S_0]$. It is a simple exercise to generalize the notation to allow service times to depend on color. Finally, setup times are iid and have mean $\mathsf{E}[R]$ and variance $\mathsf{V}[R]$, and are assumed to be independent of job service times.

A job's sojourn time in the *entire* system is built up as follows. First, we assemble jobs into batches of the same color. For simplicity, we take B constant and the same for both colors. Once B jobs of one color arrived, the batch is complete, and the *batch enters* a queue; thus we consider a queue with batches, not single jobs. After some time the batch reaches the head of the queue, and its service starts as soon as the machine becomes free. To serve a batch, the machine first performs a setup, and then processes each job individually until the batch is finished. Once a job's service is completed, it leaves the server.

In general, B should depend on the arrival rate of the job type when the arrival rates vary considerably between the famlilies.

It remains to make a quantitative model of this queueing system.

WHEN A JOB of color i arrives, the expected time for its batch to be formed is given by

[6.2.4]

$$\frac{B-1}{2\lambda_i}, \quad i \in \{r, b\}. \tag{6.2.1}$$

WHEN THE BATCH is complete, the batch joins the queue, so we next compute the average time batches spend in queue. For this we can use Sakasegawa's formula, and this formula makes us easy because we only have to find expressions for each of its components.

Note that we shift interpretation: batches (not individual items) form the queue.

The total arrival rate of jobs is $\lambda = \lambda_b + \lambda_r$. Thus, $\lambda_B = \lambda / B$ is the arrival rate of *batches*.

The expected service time of a batch is

$$\mathsf{E}[S_B] = \mathsf{E}[R] + B\,\mathsf{E}[S_0]\,. \tag{6.2.2}$$

With the batch arrival rate and expected batch service time, the load becomes $\rho = \lambda_B \mathsf{E}[S_B]$

It is essential that B is sufficiently large to ensure that $\rho < 1$. With (6.2.2) this leads to the condition that B must be larger than some minimal batch size, i.e.,

$$B > B_m = \frac{\lambda\,\mathsf{E}[R]}{1-\lambda\,\mathsf{E}[S_0]}.$$

Now that we have identified the arrival rate and service times of batches, it remains to find expressions for the scv of the batch inter-arrival times $C^2_{a,B}$ and the batch service times $C^2_{s,B}$. It's easy to derive that [6.2.5] [6.2.6]

$$C^2_{a,B} = \frac{C^2_a}{B}, \qquad C^2_{s,B} = \frac{\mathsf{V}[S_B]}{(\mathsf{E}[S_B])^2}, \tag{6.2.3}$$

where

$$\mathsf{V}[S_B] = \mathsf{V}[R] + B\,\mathsf{V}[S_0]\,.$$

Observe that we now have all components for Sakasegawa's formula.

It can useful to convert the effects of the setup time into an *effective processing time* of individual jobs. By dividing by B we see

$$\mathsf{E}[S] = \frac{\mathsf{E}[S_B]}{B} = \mathsf{E}[S_0] + \frac{\mathsf{E}[R]}{B}, \qquad \mathsf{V}[S] = \mathsf{V}[S_0] + \frac{\mathsf{V}[R]}{B}. \tag{6.2.4}$$

Note in particular that the variance of the effective service times is larger than the variance of the net processing times.

IT IS LEFT to find a rule to determine what happens to an item after it has been processed. If the job has to wait until all jobs in the batch are served, the expected time it spends at the server is $\mathsf{E}[R] + B\,\mathsf{E}[S_0]$. However, if the item can leave right after being served, the expected time at the server is [6.2.7]

$$\mathsf{E}[R] + \frac{B+1}{2}\,\mathsf{E}[S_0]\,; \tag{6.2.5}$$

the first component is the average time a job has to wait before its service starts, the second is its service time. Our model is complete!

WE CAN OBTAIN A NUMBER of important insights from the above model. Using the code from [6.2.3] we plot the sojourn time $\mathsf{E}[J_r]$ of a red job for various values of B in the figure at the right.

First, as B increases, we see a sharp decline of $\mathsf{E}[J_r]$. The reason for this is that the load ρ decreases as a function of B. Since $\mathsf{E}[W] \sim (1-\rho)^{-1}$, it is essential to stay away from critically loading the server.

When B becomes quite large, we see that the sojourn time increases linearly. This follows right away from (6.2.1) and (6.2.5), because the time to (dis)assemble batches is linear in B.

Finally, observe that the graph is not symmetric around the minimum. It is much worse to take B too small than too large. More generally, when

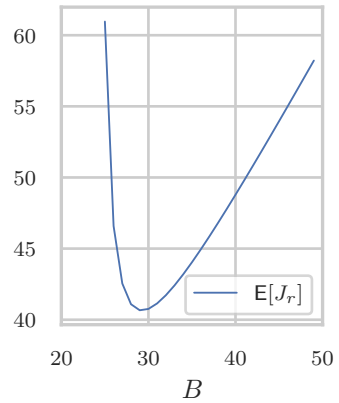


Fig. 6.2.1: The sojourn time of the red jobs as function the batch size B.

designing systems, we can use such graphs to understand how sensitive the system is to variation and measurement errors.

For the purpose of making this simple, but useful, figure, we took B and (the distribution of) S_0 the same for the red and blue jobs. Allowing for different batch sizes and net processing times per color is very simple indeed: just label the relevant symbols with and carry out the algebra.

The code I used to make Fig. 6.2.1; it's completely straightforward if you can read the maths.

```
import matplotlib.pyplot as plt

from latex_figures import fig_in_latex_format


labda = 3  # per hour
ES0 = 15.0 / 60  # hour
ER = 2.0

x = list(range(25, 50))
y = []

for B in x:
    ESe = ES0 + ER / B
    rho = labda * ESe

    # The time to form a red batch
    labda_r = 0.5
    EW_r = (B - 1) / (2 * labda_r)

    # Now the time a batch spends in queue
    Cae = 1.0
    CaB = Cae / B
    Ce = 1.0  # scv of service times
    VS0 = Ce * ES0 * ES0
    VR = 1.0 * 1.0  # Var setups is sigma squared
    VSe = B * VS0 + VR
    ESb = B * ES0 + ER
    CeB = VSe / (ESb * ESb)
    EW = (CaB + CeB) / 2 * rho / (1 - rho) * ESb

    # The time to unpack the batch, i.e., the time at the server.
    ES = ER + (B - 1) / 2 * ES0 + ES0

    total = EW_r + EW + ES
    y.append(total)


def cm_to_inch(cm):
    return cm / 2.54


plt.figure(figsize=(cm_to_inch(5), cm_to_inch(6)))
plt.xlim(20, 50)
plt.plot(x, y, label=r"$\mathsf{E}[J_r]$", lw=0.7)
plt.xlabel("$B$")
plt.legend(loc="lower right")
```

```
48  plt.tight_layout()
49  plt.savefig("../figures/setups.pdf")
```

TF 6.2.1. Jobs arrive at rate λ and are assembled into batches of size B. The average time a job waits until the batch is complete is $\mathsf{E}[W] = \frac{B-1}{2\lambda}$.

TF 6.2.2. Claim: for a $G^B/B/1$ queue, the waiting time of a job can be reduced by making the batches larger as this means that there are fewer batches in the queue.

Before dealing with technical derivations, let us first see how to apply the model.

Ex 6.2.3. Jobs arrive at $\lambda = 3$ per hour at a machine with $C_a^2 = 1$; service times are exponential with an average of 15 minutes. Assume $\lambda_r = 0.5$ per hour, hence $\lambda_b = 2.5$ per hour. Between any two batches, the machine requires a cleanup of 2 hours, with a standard deviation of 1 hour. First find the smallest batch size that can be allowed, then compute the average time a red job spends in the system in case $B = 30$ jobs.

Ex 6.2.4. Show that the average time a job has to wait to fill the batch (to which this job belongs) is given by (6.2.1).

Ex 6.2.5. Explain that the scv of the batch inter-arrival times is given by (6.2.3).

Ex 6.2.6. Show that $C_{s,B}^2$ takes the form as in (6.2.3).

Ex 6.2.7. Show that, when items can leave right after being served, the time at the server is given by (6.2.5)

6.3 SERVER ADJUSTMENTS

In Section 6.2 we study the effect of setup times between batches of jobs. There are, however, other reasons to interrupt the server from serving jobs that are in queue. For instance, when a knife in a cutting machine becomes blunt, an operator has to stop the machine to replace the knife. Another example is a medical doctor who has to make a number of unexpected phone calls between seeing two patients. All such small (maintenance) tasks can be carried out in between jobs, but it is often hard to predict when they are required. Therefore, the number of jobs served between any two tasks is no longer constant. But we know from Sakasegawa's formula that randomness in service times affects queueing times, and we also know from (6.2.4) that the batch size affects the service time. Hence, such unplanned tasks must have a negative effect on sojourn times.

Compare this to a batch of jobs.

And compare a task to a setup.

We refer to this type of interruption as a server *adjustment*. It is important to realize that setups and adjustments are *non-preemptive outages*, i.e., they occur *between* jobs, not *during* job service times. Section 6.4 deals with the latter set of outages.

In this section we develop a simple model to understand the impact of adjustments on job sojourn times; we use the same notation as in Section 6.2 and follow the same line of reasoning. With this model, we can analyze a number of trade-offs, such doing fewer, but longer adjustments, or planning adjustments instead just waiting until it becomes necessary

at an unexpected moment. With the model we can make graphs of the sojourn time as a function of adjustment rate, so that we can optimize for the adjustment rate.

[6.3.4]

LET US WRITE F_i for the Bernouilli rv that, when equal to 1, indicates that job i requires an adjustment, and, when 0, the adjustment is not necessary. We assume that $\{F_i\}$ is a set of iid rvs, distributed as the common rv F such that $\mathsf{P}\{F=1\} = p = 1 - \mathsf{P}\{F=0\}$. Thus, with constant probability p an adjustment occurs between any two jobs. As a consequence, the number of jobs served between two consequtive adjustments is a geometric rv B such that $\mathsf{E}[B] = 1/p$, and B is memoryless. We further assume that the adjustment times $\{R_i\}$ are iid rvs, distributed as the common rv R with mean $\mathsf{E}[R]$ and variance $\mathsf{V}[R]$. Finally, we assume that the net processing times $\{S_{0,i}\}$, $\{R_i\}$ and $\{F_i\}$ are independent.

In practice, we measure the average number of jobs $\mathsf{E}[B]$ served between a number of adjustments, and take $p = 1/\mathsf{E}[B]$.

TO COMPUTE $\mathsf{E}[W]$ with Sakasegawa's formula, we only have to find out how the adjustments affect the mean $\mathsf{E}[S]$ and scv C_s^2 of the effective processing time. As the adjustments have no influence on the job arrival process, λ and C_a^2 remain the same.

In the exercises we show that the average effective processing time is

$$\mathsf{E}[S] = \mathsf{E}[S_0] + p\,\mathsf{E}[R] = \mathsf{E}[S_0] + \frac{\mathsf{E}[R]}{\mathsf{E}[B]}, \tag{6.3.1}$$

and its variance is

$$\begin{aligned} \mathsf{V}[S] &= \mathsf{V}[S_0] + p\,\mathsf{V}[R] + p(1-p)(\mathsf{E}[R])^2 \\ &= \mathsf{V}[S_0] + \frac{\mathsf{V}[R]}{\mathsf{E}[B]} + (\mathsf{E}[R])^2 \frac{C_B^2}{\mathsf{E}[B]}, \end{aligned} \tag{6.3.2}$$

where C_B^2 is the scv of the runlength B. By dividing $\mathsf{V}[S]$ by $(\mathsf{E}[S])^2$ we obtainx C_S^2.

Assuming there is one server, we can now fill in Sakasegewa's formula!

BEFORE CONSIDERING SOME examples, let us compare the results of this model to those of Section 6.3. The expected effective service time is the same: an amount $\mathsf{E}[R]/\mathsf{E}[B]$ gets added to the net processing time $\mathsf{E}[S_0]$. However, the impact on the variance is different. By comparing (6.2.4) to (6.3.2) we see that the latter has an extra (positive) term. This supports our intuition: Unexpected interruptions have a larger effect on variability than expected (planned) interruptions.

The first few exercises demonstrate how to do apply the above, the rest are concerned with deriving (6.3.1) and (6.3.2).

TF 6.3.1. The number of jobs served between two consecutive adjustments follows a Poisson distribution.

TF 6.3.2. A repair shop is considering to buy a new CNC machine. There are two options available, the Yamazaki and the Haas. The Haas is less reliable and it breaks down every week on average. It can however easily be fixed by running through the control checklist, which takes 60 minutes. The Yamazaki breaks down about once a month, but it is much more complicated to fix and takes 3.5 hours. The time till a break down is memoryless

and the machines have the same service rate. Claim: The Yamazaki is better from a queueing perspective.

Ex 6.3.3. Jobs arrive as a Poisson process with rate $\lambda = 9$ per working day. The machine works two 8 hour shifts a day. Work not processed on a day is carried over to the next day. Job service times are 1.5 hours, on average, with standard deviation of 0.5 hours. Outages occur on average between 30 jobs. The average duration of an outage is 5 hours and has a standard deviation of 2 hours. Compute $\mathsf{E}[J]$.

Ex 6.3.4. In the setting of [6.3.3] we can perhaps choose to do an adjustment after every 20 jobs. For simplicity, assume that then adjustments never occur at random. Also assume that an adjustment takes less time, for instance 4.5 hour and are constant. What is $\mathsf{E}[J]$ now?

Ex 6.3.5. The easiest way to derive the expressions for $\mathsf{E}[S]$ and $\mathsf{V}[S]$ is to use Adam's and Eve's rule. Try it.

$$\mathsf{V}[Y] = \mathsf{E}[\mathsf{V}[Y|X]] + \mathsf{V}[\mathsf{E}[Y|X]].$$

6.4 SERVER FAILURES

In Sections 6.2 and 6.3 we assumed that servers are never interrupted while serving a job. However, this assumption is not always satisfied, for instance, a machine may fail in the midst of processing of a job. In this section, we develop a model to compute the influence on the mean waiting time of such *preemptive outages*, again based on Sakasegawa's formula for the $G/G/1$ queue.

There is not much point in combining the models of Section 6.2–Section 6.4 into one large model. For instance, when a machine can be setup, while a part is being repaired, there may not be time lost on the setup. All depends on the specific strategies to combine outages.

Just as in Section 6.3, we only have to derive expressions for the expectation and variance of the effective processing time S. The other components of Sakasegawa's formula are not affected.

We assume that a break down can occur at any moment during a job's service time. The repair times $\{R_i\}$ are a set of iid rvs, distributed as the common rv R and have mean $\mathsf{E}[R]$ and finite second moment $\mathsf{E}[R^2]$. Supposing that N interruptions occur during the net job service time S_0, the effective service time becomes

$$S = S_0 + S_N = S_0 + \sum_{i=1}^{N} R_i. \tag{6.4.1}$$

In an insurance context, if we interpret $\{R_i\}$ as a set of claims, then $\sum_{i=1}^{N} R_i$ is the total claim size of N claims. Likewise, in an inventory system, this sum is the total demand of N customers.

In general, N is a random number, so we need to adjust for this in the computation of the mean and variance of S.

IT IS COMMON to assume that the time between two interruptions is memoryless with mean $1/\lambda_f$, where λ_f is the *failure rate*.

Defining the mean time to fail as $m_f = 1/\lambda_f$, then the *availability* is given by

$$A := \frac{m_f}{m_f + \mathsf{E}[R]},$$

from which follows easily that

[6.4.4]

$$A = (1 + \lambda_f \mathsf{E}[R])^{-1}. \tag{6.4.2}$$

Next, because by assumption the time between two failures is $\sim \mathrm{Exp}(\lambda_f)$ so that $N \sim \mathrm{Pois}(\lambda S_0)$ if S_0 is given,

$$\mathsf{E}[N] = \mathsf{E}[\mathsf{E}[N|S_0]] = \mathsf{E}\left[\lambda_f S_0\right] = \lambda_f \mathsf{E}[S_0]. \tag{6.4.3}$$

Therefore,

$$\begin{aligned}\mathsf{E}[S] &= \mathsf{E}[S_0] + \mathsf{E}[N]\,\mathsf{E}[R] = \mathsf{E}[S_0] + \lambda_f \mathsf{E}[S_0]\,\mathsf{E}[R] \\ &= \mathsf{E}[S_0]\,(1 + \lambda_f \mathsf{E}[R]) = \mathsf{E}[S_0]\,/\,A.\end{aligned}$$

With this expression for the expectted effective servie, the load becomes

$$\rho = \lambda \mathsf{E}[S] = \lambda \frac{\mathsf{E}[S_0]}{A}.$$

Clearly, $A \in (0,1)$, hence the load increases due to failures.

WITH SOME WORK, we obtain from (6.4.1), *[6.4.5]*

$$\mathsf{E}[S] = (1 + \lambda_f \mathsf{E}[R])\,\mathsf{E}[S_0], \qquad \mathsf{V}[S] = \frac{\mathsf{V}[S_0]}{A^2} + \lambda_f \mathsf{E}\left[R^2\right]\mathsf{E}[S_0]. \tag{6.4.4}$$

By assuming the repair times are exponentially distributed, it is possible to find a neat expression for the scv of the service times *[6.4.7]*

$$C_s^2 = C_0^2 + 2A(1-A)\frac{\mathsf{E}[R]}{\mathsf{E}[S_0]}, \tag{6.4.5}$$

where C_0^2 is the scv of S_0.

TF 6.4.1. A job's normal service time, without interruptions, is given by S_0. The durations of interruptions are given by the iid rvs $\{R_i\}$ and have common mean $\mathsf{E}[R]$ and variance $\mathsf{V}[R]$. If N interruptions occur, the effective service time will then be

$$S = S_0 + \sum_{i=1}^{N} R_i.$$

Then all steps in the computation below are correct:

$$\mathsf{E}\left[\sum_{i=1}^{N} R_i\right] = \mathsf{E}\left[\sum_{n=0}^{\infty} \mathbb{1}_{N=n}\right]\mathsf{E}\left[\sum_{i=1}^{n} R_i\right] = \mathsf{E}[N]\,\mathsf{E}[R]$$

TF 6.4.2. Assume that server failures are memoryless and happen at a rate of 6 per hour. Moreover, assume that the expected amount of time to fix a failure is 18 minutes. An investment can be made which will result in a failure rate of 18 per hour and an expected amount of repair time of 6 minutes per failure. Claim: This is a good investment.

Ex 6.4.3. Suppose we have a machine with memoryless failure behavior, with a mean-time-to-fail of $m_f = 3$ hours. Regular service times are deterministic with an average of $S_0 = 10$ minutes, jobs arrive as a Poisson process with rate of $\lambda = 4$ per hour. Repair times are exponential with a mean duration of $\mathsf{E}[R] = 30$ minutes. What is the average sojourn time?

Ex 6.4.4. Derive (6.4.2).

Ex 6.4.5. Derive (6.4.4) using Adam's and Eve's law.

Ex 6.4.6. Show that

$$C_s^2 = \frac{\mathsf{V}[S]}{(\mathsf{E}[S])^2} = C_0^2 + \frac{\lambda_f \mathsf{E}[R^2] A^2}{\mathsf{E}[S_0]},$$

Ex 6.4.7. Assuming that the repair times R are exponentially distributed, show (6.4.5).

Ex 6.4.8. Without mentioning, we used the renewal reward theorem in Section 6.4 to find an expression for the availability A. How did we apply it to find (6.4.2).

6.5 $G/G/c$ QUEUES IN TANDEM

'In tandem' means 'in line', i.e., one station after the other.

Consider two $G/G/1$ stations in tandem. Suppose we have the financial means to reduce the variability of the processing times at one of the stations, but not at both. Then we like to improve the one that has the most impact on the total sojourn time in the line.

For the waiting time of the first machine, we can use Sakasegawa's formula, but to apply this to the second machine, we need $C_{a,2}^2$, i.e., the scv of the inter-arrival times at the second station. Now, noting that the output of the first machine forms the input of the second machine, it is clear that $C_{a,2}^2 = C_{d,1}^2$, where $C_{d,1}^2$ is the scv of the *inter-departure* times of *first* station.

Let us consider the inter-departure times of a $G/G/1$ queue. Suppose that the utilization ρ is very high. Then the server will seldom be idle, so that most of the inter-departure times are equal to the service times. However, if the utilization is low, the server will be idle most of the time, and the inter-departure times must be approximately equal to the inter-arrival times.

We obtain an approximation for the scv C_d^2 of the inter-departure times by interpolating between these two extremes

$$C_d^2 \approx (1-\rho^2)C_a^2 + \rho^2 C_s^2. \tag{6.5.1}$$

For the $G/G/c$, i.e., a multiserver queue, there is the generalization

$$C_d^2 \approx 1 + (1-\rho^2)(C_a^2 - 1) + \frac{\rho^2}{\sqrt{c}}(C_s^2 - 1).$$

It is simple to see that this reduces to (6.5.1) for the $G/G/1$ queue.

Combining Sakasegawa's formula with this expression provides us with a very useful insight for a line of queues. If we reduce $C_{s,1}^2$, i.e., the scv of service times at the first station, $\mathsf{E}[W_1]$ and $C_{d,1}^2$ become smaller. Since $C_{a,2}^2 = C_{d,1}^2$, $\mathsf{E}[W_2]$ becomes lower too, but also $C_{d,2}^2$, and so on. In other words, the entire chain benefits from an improvement in service variability at the first station.

In other words, try to improve from the start of the chain.

TF 6.5.1. Suppose in a tandem network of $G/G/c$ queues we can reduce C_s^2 of just one station by a factor 2. To improve the average waiting time in the entire chain, it is best to reduce $C_{s,1}^2$.

TF 6.5.2. A production system consists of 2 stations in tandem. The first station has one machine, the second has two identical machines. Machines never fail and service times are deterministic. Jobs arrive at rate 1 per hour. The machine at first station has a service time of 45 minutes per job, a machine at the second station has a service time of 80 minutes. We claim that the second station is the bottleneck.

TF 6.5.3. Consider a network with n stations in tandem. At station i, the service times S_i for all machines at that station are the same and constant; station i contains N_i machines. The number of jobs required to keep all machines busy is $N = \sum_{i=1}^{n} N_i$, and the raw processing time $T_0 = \sum_{i=1}^{n} S_i$. Thus, if the number w of allowed jobs in the system is larger than N, the number of jobs waiting somewhere in queue is $w - N$.

TF 6.5.4. We have two $M/M/1$ stations in tandem. The average queueing time for the network is given by

$$\mathsf{E}[W] = \frac{\rho_1}{1-\rho_1} + \frac{\rho_2}{1-\rho_2}.$$

Ex 6.5.5. Consider two $G/G/1$ stations in tandem. Suppose $\lambda = 2$ per hour, $C_{a,1}^2 = 2$, $C_{s,1}^2 = C_{s,2}^2 = 0.5$, and $\mathsf{E}[S_1] = 20$ minutes and $\mathsf{E}[S_2] = 25$ minutes. Compute $\mathsf{E}[J] = \mathsf{E}[J_1] + \mathsf{E}[J_2]$.

h.1.2.6. Recall that $\mathsf{E}[X] = M_X'(0)$, $\mathsf{E}\left[X^2\right] = M_X''(0)$, and $\mathsf{V}[X] = \mathsf{E}\left[X^2\right] - (\mathsf{E}[X])^2$.

h.1.2.7. Use that for independent rvs X, Y, $M_{X+Y}(t) = M_X(t)M_Y(t)$. Why is $M_{A_i}(t) = \mathsf{E}\left[e^{tA_i}\right] = \prod_{k=1}^{i} \mathsf{E}\left[e^{tX_k}\right]$?

h.1.2.8. LOTUS: $\mathsf{E}\left[\alpha^N\right] = \sum_{k=0}^{\infty} \alpha^k \mathsf{P}\{N = k\}$.

h.1.2.9. This result can be anticipated when you think about merging Poisson processes. Then, use $\{\min\{X, S\} > x\} = \{X > x\} \cap \{S > x\}$.

h.1.2.11. Think about splitting Poisson processes, and use that $\mathsf{P}\{S > X\} = \mathsf{E}[\mathsf{E}[\mathbb{1}_{S>X}|X]]$.

h.1.2.13. Dropping the dependence of N on t for the moment for notational convenience, consider the random variable

$$Y = \sum_{i=1}^{N} Z_i,$$

with $N \sim P(\lambda)$ and $Z_i \sim B(p)$ and $\{Z_i\}$ iid. Show that the moment-generating function of Y is equal to the moment-generating function of a Poisson random variable with parameter λp.

h.2.1.3. Compute the number of jobs that depart from queue 1. Subtract the used capacity for these jobs from the total capacity to get the capacity remaining for queue 2.

h.2.1.5. Modify d_k in (2.1.1) to incorporate the changed service behavior. Then, substitute d_k in the expression for L_k.

h.2.2.4. Introduce a variable $I_k \in \{0, 1\}$ to keep track of the state of the server. Then, $I_{k+1} = \mathbb{1}_{L_k \geq N} + I_k \mathbb{1}_{0<L_k<N}$ implements the N-policy.

h.2.5.7. When the machine makes n items, and each of these is faulty with probability p, then the number of faulty items is $\text{Bin}(n, p)$.

h.2.5.8. Make a queue for the new and repaired items and use [2.1.4].

h.3.1.5. Use that $L(A_k-) > 0$ means that the system contains at least one job at the time of the kth arrival, and that $A_k- < D_{k-1}$ means that job k arrives (almost surely) before job $k-1$ departs.

h.3.1.6. Make a plot of the function $t - A_{A(t)}$.

h.3.4.5. $\max\{k : A_{k-1} \leq t < A_k\} = \max\{k : A_{k-1} \leq t\}$ because the epochs A_k are ordered and separated in time (almost surely).

h.4.1.4. As a start, the function $\sin(t)$ does not have a limit as $t \to \infty$. However, the time-average $\sin(t)/t \to 0$. Now you need to make some function whose time-average does not converge, hence it should grow fast, or fluctuate wilder and wilder.

For the mathematically interested, we seek a function whose Cesàro limit does not exist.

h.4.2.4. Consider a queueing system with constant service and constant inter-arrival times.

h.4.3.5. Observe that $Y(t) = \int_0^t \mathbb{1}_{L(s)>0}\,\mathrm{d}s$ is the total amount of time the server is busy during $[0,t]$, hence $Y(t)/t$ is the fraction of time busy. Then take $T_k = D_k$ as the epochs at which to inspect $Y(t)$, and realize that $Y(D_k) = \sum_{i=1}^{k} S_i$.

h.4.3.6. For the direction of the inequalities (4.3.2), observe that t can lie half way a service interval, and $A(t) \geq D(t)$. Divide by t in (4.3.2). Then for the LHS, multiply by $A(t)/A(t)$ and take appropriate limits. Similar for the RHS but multiply by $D(t)/D(t)$.

h.4.4.5. Recall that $\lim_{t\to\infty} t^{-1}\int_0^t L_s(s)\,\mathrm{d}s$ is the time-average number of servers busy up to t.

h.4.4.6. What happens if the population increases?

h.4.5.7. Solve [4.5.6] first. Use that $\lambda \geq \delta$ always holds. Thus, when $\lambda \neq \delta$, it must be that $\lambda > \delta$. What are the consequences of this inequality; how does the queue length behave as a function of time?

h.4.6.7. Apply PASTA and (4.6.7).

h.5.1.7. Use (5.1.1), show that $M_L(s) = (1-\rho)\sum_n e^{sn}\rho^n$, then use the familiar expression of the sum over a geometric series. Similarly, $\mathsf{P}\{L \geq n\} = \sum_{k\geq n} p(k)$.

h.5.1.9. Fill in $c = 1$. Realize that this is a check on the formulas.

h.5.1.11. Use that $\sum_{i=0}^{n} x^i = (1-x^{n+1})/(1-x)$. BTW, is it necessary for this expression to be true that $|x| < 1$? What should you require for $|x|$ when you want to take the limit $n \to \infty$?

h.5.2.5. $\mathsf{E}[Q]$ follows right away from an application of Little's law. For the other quantities we need to find $\mathsf{E}[S]$. Use the expression for $\mathsf{E}[W]$ for the $M/M/1$ queue to solve for ρ. Then, since λ is known, $\mathsf{E}[S]$ follows.

h.5.2.8. Let T be the number of pattys waiting and G the number of persons waiting. There cannot be a patty and a person waiting at the same time, hence $T \cdot G = 0$. Take L as the number of jobs in an $M/M/1$ queue. If $L = 0$, then there are 3 pattys on stock, when $L = 1$ there is one patty less $T = 2, G = 0$; when $L = 2$, we have two pattys less, so $T = 1, G = 0$, when $L = 3$, there are no pattys on stock, but also no people waiting, hence $T = 0, G = 0$. When $L = 4$, we ran out of pattys and one person must be waiting, hence $T = 0, G = 1$.

With this reasoning we see a pattern: at all moments in time, $3 = T + L - G$. Now map this system to an $M/M/1$ queue.

h.5.2.9. How is the *invariant* $3 = T + L - G$ affected?

h.5.3.6. $\mathsf{P}\{B = k\} = q^{k-1}p$ with $q = 1-p$. Look up the expectation and variance of a geometric rv.

h.5.3.10. Write For 1: $\sum_{k=0}^{\infty} G(k) = \sum_{k=0}^{\infty}\sum_{m=k+1}^{\infty} \mathsf{P}\{X = m\}$, reverse the summations. Then realize that $\sum_{k=0}^{\infty} \mathbb{1}_{k<m} = m$. For 2: $\sum_{i=0}^{\infty} iG(i) = \sum_{n=0}^{\infty} \mathsf{P}\{X = n\}\sum_{i=0}^{\infty} i\,\mathbb{1}_{n\geq i+1}$,and reverse the summations.

h.5.3.11. Use [5.3.10].

h.5.3.12. Show first that

$$\mu \mathsf{E}[L] = \mu \sum_{n=0}^{\infty} n\pi(n) = \lambda \frac{\mathsf{E}[B^2]}{2} + \lambda \mathsf{E}[B]\,\mathsf{E}[L] + \lambda \frac{\mathsf{E}[B]}{2}.$$

h.5.4.6. Use the argumentation that leads to (4.3.2).

h.5.4.8. Use that $\mathsf{V}[S] = 0$ for the $M/D/1$ queue and $C_s^2 = 1$ for the $M/M/1$ queue.

h.5.4.10. Use the PASTA property. Realize that when estimating $\mathsf{E}[S_r]$ along a sample path, $S_r = 0$ for jobs that arrive at an empty system.

h.5.4.11. Use that $\sum_{n=0}^{N} \alpha^n = \frac{1-\alpha^{N+1}}{1-\alpha}$.

h.5.4.14.

$$\begin{aligned} aq^2 &= aq^2, \\ bq &= 2aq\mathsf{E}[Y] - 2aq + bq + hq\mathsf{E}[S], \\ 0 &= a\mathsf{E}\left[Y^2\right] - 2a\mathsf{E}[Y] + a + b\mathsf{E}[Y] - b + \frac{1}{2}\lambda h\mathsf{E}\left[S^2\right]. \end{aligned}$$

h.6.1.4. Think about the inter-departure time of a machine that produces items every 10 minutes. What inter-arrival times will a down-stream machine see?

h.6.1.7. What is λ? What is C_a^2 in the $G/G/1$ setting; what is it in the $M/G/1$ setting?

h.6.1.8. Let S be the processing (or service) time at the server, and S_i the service time of a type i job. Then,

$$S = \mathbb{1}_{T=1} S_1 + \mathbb{1}_{T=2} S_2.$$

h.6.2.4. An arbitrary job has to wait half the time it takes to form a batch.

h.6.3.3. Get the units right. Compute the load, and then the rest.

h.6.3.4. Realize that we now deal with setups and batch processing.

h.6.4.3. Mind to work in a consistent set of units, e.g., hours. It is easy to make mistakes.

h.6.4.4. Observe that $m_f = 1/\lambda_f$ and $m_r = \mathsf{E}[R]$.

h.6.4.5. Condition on S_0 and N.

h.6.4.6. Just realize that $\mathsf{E}[S] = \mathsf{E}[S_0]/A$, and use the above.

h.6.4.8. Define suitable moments $T_1 < T_2 < \ldots$ to inspect the system, and try to find a suitable function $Y(t)$ that captures what we want to measure.

s.1.1.1. False. The variable a changes from 8 to 17 to 10 to 7.

s.1.1.2. True.

s.1.1.3. False. Since the key 1 is not yet in the dictionary a, the code will fail.

s.1.1.4. True. Since we now work with a defaultdict, we can right away add numbers with + = even when the key 1 is not yet present in the dictionary a. As the key 1 is not present, the defaultdict will set the value to 0, and then add 3.

s.1.1.5. True. A dictionary maps a *key* to a *value*, for example a studend id (the key) to a student name (value).

s.1.2.1. False. Since $N(0,s]+N(s,t]=N(0,t]$, the LHS just says that there was one arrival during $(0,t]$. The RHS says something different, in particular because $s<t$.

s.1.2.2. True.

s.1.2.3. False, it prints 8. To see why, observe that for $i=2$, $A[i-1]=SA[1]=1$ and $X[2]=2$. Thus, for $i=2$ the RHS in the loop becomes 3. Now there is the + = symbol, implying that the 3 gets added to $A[2]$. As the latter is 1, $A[2]$ becomes $1+3=4$ after the first iteration of the while loop. In the second iteration, we therefore have that $4+3$ gets added to $A[3]=1]$, so $A[3]$ becomes 8.

s.1.2.4. True. Realize that

$$\mathsf{P}\{A_k \le t\} - \mathsf{P}\{A_{k+1} \le t\} = \frac{(\lambda t)^k}{k!} e^{-\lambda t}.$$

s.1.2.5. False, it prints 32, because the algorithms keeps adding 5 to 8 until the threshold 30 is passed. It does not print the intermediate values of `tot` because the `print(tot)` is not part of the body of the loop.

s.1.2.6.

$$M_X(t) = \mathsf{E}\left[\exp(tX)\right] = \int_0^\infty e^{tx} f(x)\,\mathrm{d}x = \int_0^\infty e^{tx}\lambda e^{-\lambda x}\,\mathrm{d}x = \frac{\lambda}{\lambda - t}.$$

This last integral only converges when $\lambda - t > 0$. Next, $M_X'(t) = \lambda/(\lambda-t)^2 \implies M_X'(0) = 1/\lambda$, $M_X''(t) = 2\lambda/(\lambda-t)^3 \implies \mathsf{E}\left[X^2\right] = 2\lambda^{-2}$. Thus, $\mathsf{V}[X] = \mathsf{E}\left[X^2\right] - (\mathsf{E}[X])^2 = \frac{2}{\lambda^2} - (\frac{1}{\lambda})^2 = \lambda^{-2}$.

s.1.2.7. Using the iid property of the $\{X_i\}$,

$$M_{A_i}(t) = \mathsf{E}\left[e^{tA_i}\right] = \mathsf{E}\left[\exp\left(t\sum_{k=1}^{i} X_k\right)\right] = \prod_{k=1}^{i} \mathsf{E}\left[e^{tX_k}\right] = \left(\frac{\lambda}{\lambda - t}\right)^i.$$

From a table of moment-generating functions it follows immediately that $A_i \sim \text{Gamma}(()i,\lambda)$, i.e., A_i is Gamma distributed.

s.1.2.8. Check the hint.

$$\sum_{k=0}^{\infty} \alpha^k \mathsf{P}\{N=k\} = \sum_{k=0}^{\infty} \alpha^k \frac{(\lambda)^k}{k!} e^{-\lambda} = e^{-\lambda} \sum_{k=0}^{\infty} \frac{(\alpha\lambda)^k}{k!} = \exp(\lambda(\alpha-1)).$$

Taking $\alpha = e^s$ in [1.2.8]:

$$M'_{N(t)}(s) = \lambda t e^s \exp(\lambda t(e^s-1)).$$

Hence $\mathsf{E}[N(t)] = M'_{N(t)}(0) = \lambda t$. Next, $M''_{N(t)}(s) = (\lambda t e^s + (\lambda t e^s)^2)\exp(\lambda t(e^s-1))$, hence $\mathsf{E}\left[(N(t))^2\right] = M''(0) = \lambda t + (\lambda t)^2$, and thus, $\mathsf{V}[N(t)] = \mathsf{E}\left[(N(t))^2\right] - (\mathsf{E}[N(t)])^2 = \lambda t + (\lambda t)^2 - (\lambda t)^2 = \lambda t$.

s.1.2.9. $\mathsf{P}\{Z>x\} = \mathsf{P}\{\min\{X,S\}>x\} = \mathsf{P}\{X>x, S>x\} = \mathsf{P}\{X>x\}\,\mathsf{P}\{S>x\} = e^{-\lambda x}e^{-\mu x}$, as X and S independent.

s.1.2.10. With the above:

$$\mathsf{P}\left\{N_\lambda(t)=1 \,\middle|\, N_\lambda(t)+N_\mu(t)=1\right\} = \frac{\mathsf{P}\{N_\lambda(t)=1\}\,\mathsf{P}\left\{N_\mu(t)=0\right\}}{\mathsf{P}\left\{N_{\lambda+\mu}(t)=1\right\}}$$
$$= \frac{\lambda t\exp(-\lambda t)\exp(-\mu t)}{((\lambda+\mu)t)\exp(-(\lambda+\mu)t)} = \frac{\lambda t\exp(-(\lambda+\mu)t)}{((\lambda+\mu)t)\exp(-(\lambda+\mu)t)} = \frac{\lambda}{\lambda+\mu}.$$

s.1.2.11. With conditioning, $\mathsf{P}\{S>X|X=t\} = \mathsf{P}\{S>t\} = e^{-\mu t}$. With the fundamental bridge and conditional expectation, $\mathsf{E}[\mathbb{1}_{S>X}|X] = e^{-\mu X}$, hence $\mathsf{E}[\mathbb{1}_{S>X}] = \mathsf{E}\left[e^{-\mu X}\right]$. But this last formula is equal to $M_X(-\mu)$, so taking $t=-\mu$ in the MGF of X we obtain $\lambda/(\lambda+\mu)$.

s.1.2.12.

$$M_{N_\lambda(t)+N_\mu(t)}(s) = M_{N_\lambda(t)}(s)\cdot M_{N_\mu(t)}(s) = \exp(\lambda t(e^s-1))\cdot\exp(\mu t(e^s-1))$$
$$= \exp((\lambda+\mu)t(e^s-1)).$$

s.1.2.13. Use the hint. Then,

$$\mathsf{E}\left[e^{sZ}\right] = e^0\,\mathsf{P}\{Z=0\} + e^s\,\mathsf{P}\{Z=1\} = (1-p)+e^s p,$$

from which

$$\mathsf{E}\left[e^{s\Sigma_{i=1}^n Z_i}\right] = \left(\mathsf{E}\left[e^{sZ}\right]\right)^n = \left(1+p(e^s-1)\right)^n,$$

where we use that the Z_i are iid. Thus, $\mathsf{E}\left[e^{sY}|N=n\right] = \mathsf{E}\left[e^{s\Sigma_{i=1}^n Z_i}\right] = \alpha^n$, where we write $\alpha = 1+p(e^s-1)$ for ease. Now with Adam's law and [1.2.8], we get

$$\mathsf{E}\left[e^{sY}\right] = \mathsf{E}\left[\alpha^N\right] = e^{\lambda(\alpha-1)} = \exp(\lambda p(e^s-1)).$$

This is the MGF of a Poisson rv with rate λp.

s.1.3.1. B (False). The ith row of A contains the first $T-1$ number of arrival times of patient i because $A[0]=0$. Hence when $T=8$, $A[:,T]$ corresponds 7th arrival time of the patients.

s.1.3.2. False.

s.1.3.3. True.

s.1.3.4. False.

s.1.3.5. True. The solution follows directly from material from Blitzstein,i.e., standard probability, and results of the previous section. Here we go. Take $U \sim \text{Unif}(0,1)$. Take $f(x) = -\frac{1}{\lambda}\log x$. Then,

$$\mathsf{P}\{f(U) \le a\} \overset{1}{=} \mathsf{P}\{U \ge f^{-1}(a)\} \overset{2}{=} 1 - f^{-1}(a) \overset{3}{=} 1 - e^{-\lambda a},$$

where 1 follows from the fact that f is strictly decreasing, 2 from the general property of the uniform distribution $\mathsf{P}\{U \ge b\} = 1-b$, and 3 because $f^{-1}(y) = e^{-\lambda y}$. Thus, if we define $X := f(U)$, then $X = -\frac{1}{\lambda}\log U \sim \text{Exp}(\lambda)$. In other words, the rv $X = -\frac{1}{\lambda}\log U$ is exponentially distributed.

Next, define

$$N(t) = \inf\{j : A_j \le t < A_{j+1}\},$$

in words, $N(t)$ is the index of the last arrival before or equal t. Moreover, when $\{U_i\}$ is a sequence of iid uniform rvs on $(0,1)$ then the $X_i = -\frac{1}{\lambda}\log U_i$ are iid $\sim \text{Exp}(\lambda)$. Since $A_k = A_{k-1} + X_k$, we have that the event

$$\begin{aligned}\{N(t) = k\} &= \{A_k \le t < A_{k+1}\} \\ &= \left\{\sum_i^k X_i \le t < \sum_i^{k+1} X_i\right\} \\ &= \left\{-\frac{1}{\lambda}\sum_i^k \log U_i \le t < -\frac{1}{\lambda}\sum_i^{k+1}\log U_i\right\}.\end{aligned}$$

Now taking $t = 1$, we arrive at the equality:

$$\{N(1) = k\} = \left\{-\frac{1}{\lambda}\sum_i^k \log U_i \le 1 < -\frac{1}{\lambda}\sum_i^{k+1}\log U_i\right\}.$$

Since $-\frac{1}{\lambda}\log U \sim \text{Exp}(\lambda)$, the sum of k such rvs is $\text{Gamma}(k,\lambda)$, and, by (1.2.3) and (1.2.4),

$$\mathsf{P}\{N(1) = k\} = \mathsf{P}\left\{-\frac{1}{\lambda}\sum_i^k \log U_i \le 1 < -\frac{1}{\lambda}\sum_i^{k+1}\log U_i\right\} = \frac{\lambda^k}{k!}e^{-\lambda}.$$

In fact, this is exactly what the next algorithm implements: it searches for the smallest index of a sum of exp rvs suc that the threshold of λ exceded.

```
def Pois(labda):
    R = 0
    T = - np.log(uniform()) / labda
    while T < 1:
        R += 1
        T += -np.log(uniform()) / labda
    return R
```

However, note that we have to divide every rv `uniform()` by λ. But this is numerically wasteful. We can just as well multiply the threshold by λ, just once, and save the division by λ. And this is the algorithm of the question.

Realize that each uniform rv in the while loop must be a new drawing (i.e., realization of a rv). If not, we keep on adding the same number, which is wrong.

s.1.4.1. False.

s.1.4.2. False. The line `c[i, j] += p` is wrong. Check the real code to see the difference.

Here is a possible variation on this question. Claim: if we apply this function to two independent rvs X and Y and take `f(i,j) = i/j`, we get a `RV` that represents $Z = X/Y$.

```
def compose_function(f: Callable[[numeric, numeric], float], X: RV, Y: RV) -> RV:
    """Make the rv f(X, Y) for the independent rvs X and Y."""
    c: defaultdict[float, float] = defaultdict(float)
    for i in X.support():
        for j in Y.support():
            p = X.pmf(i) * Y.pmf(j)
            c[f(i, j)] += p
    return RV(c)
This one is True, but the code is less efficient than the real code.

Yet another variation is to write ~p = X.pmf(i) / Y.pmf(j)~. This is of course completely wrong.
```

s.1.4.3. False: not true for non continuous CDFs.

Please relate this question to the method `rvs()` (and the explanation above this code)that shows how to generate random numbers distributed according to a given cdf. In particular you should understand that the inverse of the cdf F^{-1} is relevant.

s.1.4.4. False. Check the index of the second summation.

s.1.4.5. Type on the Google prompt: 'realpython defaultdict', click on the link that appears. Then read and think hard. Recall, coding involves intellectual effort.

s.2.1.1. True.

s.2.1.2. True.

s.2.1.3. True.

s.2.1.4. True. Queue 2 minimally needs $c_k^2 = \min\{L_{k-1}^2, r^2\}$, and with this,

$$d_k^1 = \min\{L_{k-1}^1, c_k - c_k^2\}, \qquad d_k^2 = \min\{L_{k-1}^2, c_k - d_k^1\}.$$

s.2.1.5.

$$\begin{aligned} d_k &= \min\{L_{k-1} + a_k, c_k\}, \\ L_k &= L_{k-1} + a_k - d_k = L_{k-1} + a_k - \min\{L_{k-1} + a_k, c_k\} \\ &\overset{1}{=} -\min\{c_k - L_{k-1} - a_k, 0\} \overset{2}{=} \max\{L_{k-1} + a_k - c_k, 0\}, \end{aligned}$$

where 1 uses that $x - \min\{x, y\} = -\min\{0, y - x\}$ and 2 that $-\min\{x, 0\} = \max\{-x, 0\}$.

s.2.2.1. False. We can also block jobs.

s.2.2.2. False. Since two jobs arrive, in expectation, per period, and one leaves, the queue length increases with a speed of 1 job per period (again in expectation). After 10000 periods, the system must contain about 10000 jobs. As the Poisson disctribution has a small relative variability (if $X \sim \text{Pois}(\lambda)$, then $\mathsf{V}[X]/(\mathsf{E}[X])^2 = 1/\lambda$) the probability that $L_{10000} < 100$ is exceedingly small.

s.2.2.3. It is false. The next line should replace the code for $c2$.

```
c2 = c[k] - max(r1, d1)
```

The problem with the code of the question is that the reserved capacity for the first queue is not used in the computation of the capacity available for the second queue.

s.2.2.4. Think about the hint. Understanding the state variable I_k is the hardest part.

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


rng = np.random.default_rng(3)

num_weeks = 40
a = rng.poisson(1, size=num_weeks)
c = rng.poisson(5, size=num_weeks)
N = 20

L = np.zeros(num_weeks)
I = np.zeros(num_weeks)  # on or not

for k in range(1, num_weeks):
    I[k] = (L[k - 1] >= N) + I[k - 1] * (0 < L[k - 1] < N)
    c[k] *= I[k]
    d = min(L[k - 1] + a[k], c[k])
    L[k] = L[k - 1] + a[k] - d


xx = range(num_weeks)
plt.step(xx, L, where="pre")
plt.step(xx, N * I, ":", where="pre")
plt.savefig("N-policy.pdf")
```

Note that when arrivals cannot be served in the period in which they arrive, it might be that the system is never empty. For instance, if $a_k \sim$ Unif4, 10, then $L_k \geq a_k \geq 4$ for all k.

s.2.2.5. The last line is the next code most interesting. Ask chatgpt to explain it to you if you find it hard (I did it, to check. The answer is real good.)

```
beta = 0.5
K = 50
h = 1


cost = beta * I.sum()
cost += h * L.sum()
cost += K * (I[1:] > I[:-1])
```

s.2.3.1. False, they are very similar.

s.2.3.2. False, it's the inventory position.

s.2.3.3. True.

s.2.3.4. True.

s.2.3.5. In a queueing system, an item is not produced before the demand for it arrives. This is a cost effective way to produce when items are very expensive, hence holding cost is high, or customer specific. Besides, services (which are not products) cannot be put on stock. For instance, keeping an invetory of haircuts seems impossible.

Note that in a queueing system the server has a finite capacity. In the inventory models above, the production capacity is assumed infinite.

A basestock model with $s = -1$ is appropriate. When a customer arrives, the server switches on, and keeps on producing until there are no further jobs in queue. Note also that a job in queue is the same as a customer in backlog.

s.2.3.6. We want all service levels to be zero. If $h \gg b$ for some inventory system, the cost structure expresses that we dislike stock on hand. In other words, it expresses that we want produce according to MTO.

s.2.3.7. Note the differences between these schemes. In the (s,S) the order level is fixed, but in case of stochastic demand, the specific period in which an order is triggered is not fixed. The order quantity (which is not the same as the order-up-to level) may also differ from period to period. In the (Q,r) policy, the order quantity is fixed, but not the period. The (T,S) policy fixes the period, but leaves the order quantity free. The (s,S) policy must be ablle to achieve the lowest average cost, because it is the least constrained of the policies.

s.2.3.8. Since at the start of period k, only the on-hand inventory I_k can be used to serve the demand D_k, the rules for a loss system become

$$\begin{aligned} D'_k &= \min\{I_k, D_k\} && \text{accepted demand} \\ L_k &= D_k - D'_k && \text{lost demand} \\ P'_k &= P_k - D'_k, && \text{subtract the accepted demand} \\ I'_k &= I_k - D'_k, && \text{subtract the accepted demand.} \end{aligned}$$

And now use the standard rules to compute Q_{k+1}, and update P_{k+1} and I_{k+1} accordingly.

s.2.4.1. The (Q,r) rule with $Q = 10$ and $r = s = 20$.

s.2.5.1. False. The condition does not exclude that the arrival rate is the service rate, for which the queue is not stable.

s.2.5.2. True. However a bit misleading, it is true for all values of t.

s.2.5.3. False. The variance of the time required to pay has an impact on the queue length too.

s.2.5.4. False.

s.2.5.5. True.

s.2.5.6. Here is the python code.

```
import numpy as np

rng = np.random.default_rng(3)
a = rng.integers(3, 8, size=100)
c = 2 * np.ones_like(a)
L = np.zeros_like(a)
l = np.zeros_like(a)  # store the lost jobs

K = 8  # loss level

for k in range(1, len(a)):
    d = min(L[k - 1], c[k])
    Lp = L[k - 1] + a[k] - d  #  without loss
    L[k] = min(Lp, K)  #  chop off at K
    l[k] = Lp - L[k]  #  lost

print(sum(l) / sum(a))  # fraction lost.
```

s.2.5.7. Here is the code.

```
import numpy as np

rng = np.random.default_rng(3)
a = rng.integers(3, 8, size=100)
c = 7 * np.ones_like(a)
L = np.zeros_like(a)
d = np.zeros_like(a)
p = 0.1

for k in range(1, len(a)):
    produced = min(L[k - 1], c[k])
    faulty = rng.binomial(produced, p)
    d[k] = produced - faulty
    L[k] = L[k - 1] + a[k] - d[k]

print(L.mean())
```

Observe that faulty items do not leave the system.

An interesting challenge: can you use these recursions to *prove* that the long-run average service capacity $n^{-1}\sum_{i=1}^{n} c_i$ must be larger than $\gamma/(1-p)$, where $\gamma = \lim_{n\to\infty} n^{-1}\sum_{k=1}^{n} a_k$ is the arrival rate of new jobs?

s.2.5.8. In code:

```
a = [0, 4, 8, 2, 1]
c = [3] * len(a)
L_R = [0] * len(a)  # repair jobs
d_R = [0] * len(a)  # departing repair jobs
L_N = [0] * len(a)  # new jobs
d_N = [0] * len(a)  # departing new jobs
```

```
7  p = 0.2
8
9  L_R[0] = 2
10 L_N[0] = 8
11
12 for k in range(1, len(a)):
13     l = L_R[k - 1]
14     if l % 2 == 1:
15         l -= 1
16     d_R[k] = min(l, 2 * c[k])
17     c_N = c[k] - d_R[k] / 2  # capacity left for new jobs
18     d_N[k] = min(L_N[k - 1], c_N)
19     a_R = int(p * d_N[k] + 0.5)  # rounding
20     a_N = a[k]
21     L_R[k] = L_R[k - 1] + a_R - d_R[k]
22     L_N[k] = L_N[k - 1] + a_N - d_N[k]
23
24 print(L_R)
25 print(L_N)
```

We need a trick to get around the division by 2 in line 17 (because if $L_R[k]$ is odd we might end up with 0.5). For this, we should check whether $L_R[k]$ is odd or not. If so, it must be at least be 1, and then we can subtract 1 to make l even. This l can be used to compute the number of departures.

The above code is easily converted to maths. Only lines 13 to 15 might be a bit problematic, but that is easy too. Whe writing line 16 like this,

$$d_{R,k} = 2\min\{\lfloor L_{R,k-1}/2 \rfloor, c_k\}, \tag{6.5.2}$$

we obtain the effect of lines 13 to 16 in one step.

Here is a general observation: the number of ways to organize the repairs is countless. Do we serve them with priority or not, do we bin them and make new items instead, do we do the repairs on another, separate, station? Such differences need to be reflected in the simulation model.

s.2.5.9. Using [2.2.3] with $d_k^1 = \min\{L_{k-1}^1, c_k^1, M - L_{k-1}^2\}$ is nearly correct. But, what if $L_{k-1}^2 > M$ for the first few periods? Therefore, take instead $d_k^1 = \min\{L_{k-1}^1, c_k^1, [M - L_{k-1}^2]^+\}\}$.

```
1  a1 = [0, 2, 3, 8, 0, 9]
2  c1 = [3] * len(a1)
3  c2 = [2] * len(a1)
4  L1 = [0] * len(a1)
5  L2 = [0] * len(a1)
6
7  M2 = 10
8
9  if L2[0] > M2:
10     print("The starting value of L2 is too large")
11     exit(0)
12
13 for k in range(1, len(a1)):
14     d1 = min(L1[k], c1[k], M2 - L2[k - 1])
```

```
    L1[k] = L1[k - 1] + a1[k] - d1
    d2 = min(L2[k], c2[k])
    L2[k] = L2[k - 1] + d1 - d2
```

s.2.5.10. The idea is like this. The dictionary `L_count` counts the number of jobs that see 0, 1, and so on, in the system.

Here is the code. As an aside, it was hard to make it this simple. (See the web what a defaultdict does.)

```
from collections import defaultdict

L_count = defaultdict(int)

a = [0, 2, 5, 1, 2]
c = [0, 1, 1, 1, 1]

d = [0] * len(a)
L = [0] * len(a)

for k in range(1, len(a)):
    d[k] = min(L[k - 1], c[k])
    L[k] = L[k - 1] + a[k] - d[k]
    for i in range(a[k]):
        L_count[L[k - 1] - d[k] + i] += 1


# normalize
tot = sum(L_count.values())
L_dist = {k: v / tot for k, v in L_count.items()}

print(L_count)
print(L_dist)
```

s.3.1.1. True. As simple variation: Claim: with our notation, the following mapping is correct:

$$A(t) : \mathbb{R} \to \mathbb{N}, \quad \text{time (real number) to number of jobs (integer).}$$

s.3.1.2. False. Suppose $A_3 = 10$ and $A_4 = 20$. Take $t = 15$. Then $\min\{k : A_k \geq 15\} = 4$ since $A_3 < t = 15 < A_4$. However, $\max\{k : A_k \leq t\} = 3$. And, indeed, at time $t = 15$, 3 jobs arrived, not 4. Here is a simple variation: Claim: the number of arrivals $A(t)$ during $[0, t]$ can be defined as $\min\{k : A_k > t\}$?

s.3.1.3. False.

s.3.1.4. The first part of the claim is False, the other True, hence the combination is False.

$A(t)$ is the number of arrivals during $[0, t]$. Suppose that $A(t) = n$. This nth job arrived at time A_n. Thus, $A_{A(t)}$ is the arrival time of the last job that arrived before or at time t.

In a similar vein, A_n is the arrival time of the nth job. Thus, the number of arrivals up to time A_n, i.e., $A(A_n)$, must be n.

s.3.1.5. In a sense, the claim is evident, for, if the system contains a job when job k arrives, it cannot be empty. But if it is not empty, then at least the last job that arrived before job k, i.e., job $k-1$, must still be in the system. That is, $D_{k-1} \geq A_k$. A more formal proof proceeds along the following lines. Using that $A(A_k-) = k-1 = D(D_{k-1})$,

$$L(A_k-) > 0 \iff A(A_k-) > D(A_k-) \iff A(A_k-) \geq D(A_k)$$
$$\iff k-1 \geq D(A_k) \iff D(D_{k-1}) \geq D(A_k) \iff D_{k-1} \geq A_k,$$

where the last relation follows from the fact that $D(t)$ is a counting process, hence monotone non-decreasing.

s.3.1.6. Suppose that $A_{n+1} = A_n + \epsilon$, i.e., the arrival time A_{n+1} is just a tiny bit larger than A_n. Then job $n+1$ would have to wait $J_n - \epsilon$ before it can get access to the server. Now, recall from a previous exercise that if $A(t) = n$, then A_n is the arrival time of the nth job, so that the function $A_{A(t)}$ provides us with arrival times as a function of t. Between arrival moments, the virtual waiting time decreases with slope 1, until it hits 0.

s.3.2.1. False. All statements are true. The first and second statement are evident. For the third, K is the capacity of the system, not the queue capacity. Hence, the queue cannot contain more than $K-c$ jobs, and since $K-c < K$, the third statement is true.

s.3.2.2. False. Service times are iid and exponential.

s.3.2.3. False. The service for the business customers works as a priority queue. Therefore, one of the $c+1$ servers, as perceived by the economy customers, works differently.

s.3.3.1. False. Very sneaky, but the correct quantity is `D[1:].mean()`.

s.3.3.2. True.

s.3.3.3. False. See the main text. It neglects the dependence between A_k and D_{k-1}.

s.3.4.1. True.

s.3.4.2. False, see the definition elsewhere in the book.

s.3.4.3. False. It prints the entry with 'Zeno', because 1.3 is the smallest of the first elements of the tuples in the heap.

s.3.4.4. First compute the sum of the sojourn times of all jobs that departed before t; in a formula $\sum_{k=1}^{n} J_k \mathbb{1}_{D_k \leq t}$. Then add to this the times of the jobs arrived before t but that have yet departed: $\sum_{k=1}^{n} (t - A_k) \mathbb{1}_{A_k \leq t < D_k}$. Divide the total by t.

s.3.4.5. See the hint, and check out the definition of $A(t)$.

$$\max\{k : A_{k-1} \leq t < A_k\} = \max\{A_{k-1} \leq t\} = \sum_{k=1}^{\infty} \mathbb{1}_{A_{k-1} \leq t} = \sum_{k=0}^{\infty} \mathbb{1}_{A_k \leq t} = 1 + A(t).$$

s.3.5.1. False. Compare with the example code.

s.3.5.2. False. Compare with the correct code.

s.3.5.3. True. Compare with the correct code.

An interesting variation. Claim: the job keeps a statistic of the number of free servers as seen just prior to arrival. This claim is False, because the code first pops a server from the pool of free servers, and then sets the `job.free_servers` attribute.

s.3.5.4. True.

s.3.5.6. Give this string to ChatGPT: 'What is the advantage of a python deque over a heapq. Discuss the algorithmic efficiency too.'

s.4.1.1. False. In any reasonable queueueing system this limit is ∞.

s.4.1.2. False.

s.4.1.3. False: δ is not μ

s.4.1.4. If $A(t) = 3t^2$, then clearly $A(t)/t = 3t$. This does not converge to a limit.

Another example, let the arrival rate $\lambda(t)$ be given as follows:

$$\lambda(t) = \begin{cases} 1 & \text{if } 2^{2k} \le t < 2^{2k+1} \\ 0 & \text{if } 2^{2k+1} \le t < 2^{2(k+1)}, \end{cases}$$

for $k = 0, 1, 2, \ldots$. Let $A(t) = \lambda(t)t$. Then $A(t)/t$ does not have a limit. Of course, these examples are quite pathological, and are not representable for 'real life cases'. (Although this is also quite vague: what is actually a real-life case?)

s.4.2.1. False.

$$\frac{1}{t}\sum_{k=1}^{A(t)} \mathbb{1}_{W_k \le x} = \frac{A(t)}{t}\frac{1}{A(t)}\sum_{k=1}^{A(t)} \mathbb{1}_{W_k \le x} \to \lambda \mathsf{P}\{W \le x\}.$$

Moreover, the last w in the equation in the truefalse is also wrong, and should be an x.

s.4.2.2. True.

s.4.2.3. True.

s.4.2.4. Take $L(0) = 0$, $X_k = 10$ and $S_k = 10 - \epsilon$ for some tiny $\epsilon > 0$. Then $L(t) = 1$ nearly all the time. In fact, $\lim_{t\to\infty} t^{-1}\int_0^t L(t)\,\mathrm{d}t = 1 - \epsilon/10$. However, $L(A_k-) = 0$ for all k.

s.4.2.5.

$$\mathsf{E}[L] = \lim_{t\to\infty}\frac{1}{t}\int_0^t L(s)\,\mathrm{d}s \neq \lim_{t\to\infty}\frac{L(t)}{t}.$$

If $L(t) = 1$ for all t, $\mathsf{E}[L] = 1$, but $L(t)/t \to 0$.

s.4.3.1. True. The cost increment per failure is c. Thus, this c takes the role of the X in the renewal reward theorem. The time between the failures is $\lambda = 1/\mathsf{E}[X]$, where X is the (common) interarrival time between two failures. Therefore $Y = c\lambda = c/\mathsf{E}[X]$.

s.4.3.2. True. See [4.3.7].

s.4.3.3. True.

s.4.3.4. False.

s.4.3.5. Take the hint serious. Then, as $k/T_k \to \delta$, it follows from the renewal reward theorem that $Y(t)/t \to \delta \mathsf{E}[S]$. By rate-stability, $\delta = \lambda$, so that $\rho = \lambda \mathsf{E}[S] = \delta \mathsf{E}[S] = Y$, from which the claim follows.

s.4.3.6. For the LHS, as $A(t) \to \infty$ as $t \to \infty$,

$$\lim_{t\to\infty} \frac{1}{t} \sum_{k=1}^{A(t)} S_k = \lim_{t\to\infty} \frac{A(t)}{t} \frac{1}{A(t)} \sum_{k=1}^{A(t)} S_k = \lim_{t\to\infty} \frac{A(t)}{t} \cdot \lim_{t\to\infty} \frac{1}{A(t)} \sum_{k=1}^{A(t)} S_k = \lambda \mathsf{E}[S].$$

Apply similar limits to the RHS. The limit of the middle term gives the fraction of time the server is busy.

s.4.3.7. Take $Y(t) = \int_0^t \mathbb{1}_{L_s(s)=1} \mathrm{d}t$. Then, when U_k is the kth busy time and I_k the kth idle time, let $X_k = U_k$ and $T_k = U_k + I_k$. Then $Y(t)/t \to \rho$, $X_k/k \to \mathsf{E}[U]$, and $(I_k + U_k)/k \to \mathsf{E}[I] + \mathsf{E}[U] = \lambda^{-1}$, i.e., the inverse of the λ we use in the renewal-reward theorem.

s.4.4.1. False.

s.4.4.2. False. In the first place, the W_k should be J_k. And even after replacing the W_k by the J_k, it only holds at times T at which the system is empty.

s.4.4.3. True, [4.4.5].

s.4.4.4. False.

s.4.4.5. Since the queue is stable by assumption, jobs depart at rate λ, hence any job enters and leaves the box that contains the c servers. The average time a job spends in the box is $\mathsf{E}[S]$. By Little's law, the average number in the box is $\mathsf{E}[L_s] = \lambda \mathsf{E}[S]$.

s.4.4.6. Let's make a very simple model to clarify the problem. We start with an initial population size of A. Then assume that each and every individual dies after exactly 80 years (and does not live a day longer). Moreover, assume that the population increases by 1% per year. Then, after 79 years, the size of the population is $A\sum_{k=0}^{79}(1.01)^k$. If we assume (as we do when we apply Little's law) that each year contains the same amount of people, then the number of people per year equals:

$$\frac{A}{80} \sum_{k=0}^{79} (1.01)^k = \frac{A}{80} \frac{(1.01)^{80}}{1.01 - 1} = 1.5A.$$

However, after 80 years, just A people die, not $1.5A$. Interestingly, $4000/1.5 = 2700$, which is quite a bit closer to 2900 per week.

In conclusion, we should apply Little's law with caution; in particular, when we just consider stretches of time in which we know that the population increases (or decreases) on average, Little's law does not hold.

s.4.5.1. False. In the $G/M/1$ queue jobs don't arrive as a Poisson process.

s.4.5.2. True.

s.4.5.3. True.

s.4.5.4. False, for various reasons. First we should use $L(A_k-)$ instead of $L(A_k)$ to count the number of arrivals seen *upon arrival*. Seccond, α counts the fraction of arrivals that see the number of jobs at the server (not in the system). By PASTA this must be the load, which is ρ; $1-\rho$ is the fraction of time the server is idle.

BTW, here is another way to write α:

$$\alpha = \lim_{n\to\infty} \frac{1}{n} \sum_{k=1}^{n} \mathbb{1}_{L(A_k-)>0} = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathbb{1}_{L(s)>0}\, \mathrm{d}s.$$

s.4.5.6. If $\lambda > \delta$, then $L(t) \to \infty$. But then there must be a last time, s say, that $L(s) = n+1$, and $L(t) > n+1$ for all $t > s$. Hence, after time s there will never be a job that will see the system with n jobs. Thus $A(n,t) = A(n,s)$ for all $t > s$ (recall, $A(n,t)$ can only change at time t, say, when a job arriving at t sees n in the system. But since $L(t) > n+1$ for $t > 2$, this will never happen after ts.) As $A(n,t)$ remains finite, $\lim_{t\to\infty} A(n,t)/t = 0$.

s.4.5.7. The assumptions lead us to conclude that $\lambda > \delta$. As a consequence, the queue length must increase in the long run (jobs come in faster than they leave). Therefore, $A(n,t)/t \to 0$ for all n, and also $D(n,t)/t \to 0$. Consequently, $\pi(n) = \delta(n) = 0$, which is the only sensible reconciliation with (4.6.6).

s.4.5.8. By the memoryless property of the (exponential) distributed service times of the $M/M/1$ queue, the duration of a job in service, if any, is $\mathrm{Exp}(\mu)$ also at an arrival moment. Therefore, at an arrival moment, all jobs in the system (whether in service or not) have the same expected duration. Hence, the expected time to spend in queue is the expected number of jobs in the system times the expected service time of each job, i.e., $\mathsf{E}[W] = \mathsf{E}[L]\,\mathsf{E}[S]$. Note that we use PASTA to see that the expected number of jobs in the system at an arrival is $\mathsf{E}[L]$. For the $M/G/1$ queue, the job in service (if any) does not have the same distribution as a job in queue. Hence, for the $M/G/1$ queue, the *expected time in queue is not* $\mathsf{E}[L]\,\mathsf{E}[S]$.

s.4.6.1. True.

s.4.6.2. True.

s.4.6.3. False. The figure sketches the $M/M^2/1/3$ queue.

s.4.6.4. False. Since service times are constant in the $M/D/1$ queue, hence not memoryless, $\mu \neq \mu(n+1)$ for the $M/D/1$ queue.x

s.4.6.5. 1. $A_k = 2(k-1)$, $k = 1,2,\ldots$, as jobs arrive at $t = 0,2,4,\ldots$ Thus, $A(t) = 1 + \lfloor t/2 \rfloor$ for $t \geq 0$. (BTW, $A(t)/t \to 1/2$ as $t \to \infty$.) We also know that $L(s) = 1$ if $s \in [2k, 2k+1)$ and $L(s) = 0$ for $s \in [2k-1, 2k)$ for $k = 0,1,2,\ldots$. Thus, $L(A_k-) = L(2k-) = 0$. Therefore, $A(0,t) = A(t)$ for $t \geq 0$, and $A(n,t) = 0$ for $n \geq 1$.

2. All arrivals see an empty system. Hence, $A(0,t)/A(t) \approx (t/2)/(t/2) = 1$, and $A(n,t) = 0$ for $n > 0$. Thus, $\pi(0) = \lim_{t\to\infty} A(0,t)/A(t) = 1$ and $\pi(n) = 0$ for $n > 0$. Recall from the other exercises that $p(0) = 1/2$. Hence, statistics as obtained via time averages are not necessarily the same as statistics obtained at arrival moments (or any other point process).

3. $\lambda = \lim_{t\to\infty} A(t)/t = 1/2$. $\lambda(0) = 1$, $p(0) = 1/2$, and $\pi(0) = 1$. Hence,

$$\lambda\pi(0) = \lambda(0)p(0) \implies \frac{1}{2} \times 1 = 1 \times \frac{1}{2}.$$

   For $n > 0$ it's easy, everything is 0.

4. Observe that the system never contains more than 1 job. Hence, $Y(n,t) = 0$ for all $n \geq 2$. Then we see that $Y(1,t) = \int_0^t \mathbb{1}_{L(s)=1}\,\mathrm{d}s$. Now observe that for our queueing system $L(s) = 1$ for $s \in [0,1)$, $L(s) = 0$ for $s \in [1,2)$, $L(s) = 1$ for $s \in [2,3)$, and so on. Thus, when $t < 1$, $Y(1,t) = \int_0^t \mathbb{1}_{L(s)=1}\,\mathrm{d}s = \int_0^t 1\,\mathrm{d}s = t$. When $t \in [1,2)$,

$$L(t) = 0 \implies \mathbb{1}_{L(t)=0} \implies Y(1,t) \text{ does not change.}$$

   Continuing to $[2,3)$ and so on gives

$$Y(1,t) = \begin{cases} t & t \in [0,1), \\ 1 & t \in [1,2), \\ 1 + (t-2) & t \in [2,3), \\ 2 + (t-4) & t \in [4,5), \end{cases}$$

   and so on. With this, we see that the general formula must be

$$Y(1,t) = \begin{cases} \lfloor t/2 \rfloor + t - \lfloor t \rfloor, & \text{if } \lfloor t \rfloor \text{ is even}, \\ 1 + \lfloor t/2 \rfloor, & \text{if } \lfloor t \rfloor \text{ is odd}. \end{cases}$$

   Since $Y(n,t) = 0$ for all $n \geq 2$, $L(s) = 1$ or $L(s) = 0$ for all s, therefore,

$$Y(0,t) = t - Y(1,t).$$

5.

$$\begin{aligned} \lambda(0) &\approx \frac{A(0,t)}{Y(0,t)} \approx \frac{t/2}{t/2} = 1, \\ \lambda(1) &\approx \frac{A(1,t)}{Y(1,t)} \approx \frac{0}{t/2} = 0, \\ p(0) &\approx \frac{Y(0,t)}{t} \approx \frac{t/2}{t} = \frac{1}{2}, \\ p(1) &\approx \frac{Y(1,t)}{t} \approx \frac{t/2}{t} = \frac{1}{2}. \end{aligned}$$

   For the rest $\lambda(n) = 0$, and $p(n) = 0$, for $n \geq 2$.

6. $D(0,t) = \sum_{k=1}^{\infty} \mathbb{1}_{D_k \leq t, L(D_k)=0}$. From the graph of $\{L(s)\}$ we see that all jobs leave an empty system behind. Thus, $D(0,t) \approx t/2$, and $D(n,t) = 0$ for $n \geq 1$. With this, $D(0,t)/Y(1,t) \sim (t/2)/(t/2) = 1$, and so,

$$\mu(1) = \lim_{t\to\infty} \frac{D(0,t)}{Y(1,t)} = 1,$$

   and $\mu(n) = 0$ for $n \geq 2$.

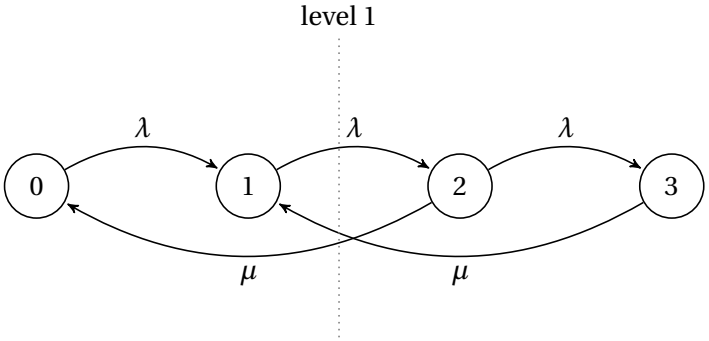
7. $\lambda(0)p(0) = 1 \cdot 1/2 = 1/2$, $\lambda(n)p(n) = 0$ for $n > 1$, as $\lambda(n) = 0$ for $n > 0$. Next, $\mu(1) = 1$, hence $\mu(1)p(1) = 1 \cdot 1/2 = 1/2$. Moreover, $\mu(n) = 0$ for $n \geq 2$. Clearly, for all n we have $\lambda(n)p(n) = \mu(n+1)p(n+1)$.

s.4.6.6. Noting that $L(s) = \sum_{n=0}^{\infty} n \, \mathbb{1}_{L(s)=n}$, we see that

$$\frac{1}{t}\int_0^t L(s)\,\mathrm{d}s = \frac{1}{t}\int_0^t \left(\sum_{n=0}^{\infty} n\,\mathbb{1}_{L(s)=n}\right)\mathrm{d}s = \sum_{n=0}^{\infty}\frac{n}{t}\int_0^t \mathbb{1}_{L(s)=n}\,\mathrm{d}s = \sum_{n=0}^{\infty} n\frac{Y(n,t}{t}$$

Taking the limit $t \to \infty$ and applying (4.5.2) (and reversing the limit and the summation, which we assume is ok here) we obtain that $\mathsf{E}[L] = \sum_n np(n)$.

s.4.6.7. Use the references in the hint to see that $\rho = \sum_{i=1}^{\infty} p(i) = \sum_{i=1}^{\infty} \pi(i) = \sum_{i=1}^{\infty} \delta(i)$.



s.4.6.8.

With level-crossing:

$$\begin{aligned}
&\lambda p(0) = \mu p(2), \quad \text{the level between 0 and 1,}\\
&\lambda p(1) = \mu p(2) + \mu p(3), \quad \text{see level 1,}\\
&\lambda p(2) = \mu p(3), \quad \text{the level between 2 and 3.}
\end{aligned}$$

Solving this in terms of $p(0)$ gives $p(2) = \rho p(0)$, $p(3) = \rho p(2) = \rho^2 p(0)$, and

$$\lambda p(1) = \mu(p(2) + p(3)) = \mu(\rho + \rho^2)p(0) = (\lambda + \lambda^2/\mu)p(0),$$

hence $p(1) = p(0)(\mu + \lambda)/\mu$. For the final answer, use the normalization constraint $p(0) + p(1) + p(2) = 1$. (As this is simple, we skip it.)

s.4.6.9.

$$\lim_{t\to\infty}\frac{I(n,t)}{t} = \lim_{t\to\infty}\frac{A(n-1,t)}{t} + \lim_{t\to\infty}\frac{D(n,t)}{t} = \lambda(n-1)p(n-1) + \mu(n+1)p(n+1)$$

and

$$\lim_{t\to\infty}\frac{O(n,t)}{t} = \lim_{t\to\infty}\frac{A(n,t)}{t} + \lim_{t\to\infty}\frac{D(n-1,t)}{t} = \lambda(n)p(n) + \mu(n)p(n)$$

s.5.1.1. False. In this queueing model, the arrival rate $\lambda(n)$ can depend on the number of jobs in queue. If this is the case, the interarrival times can still be independent, but not identically distributed.

s.5.1.2. True. See [5.1.7].

s.5.1.3. False, [1.2.12]

s.5.1.4. True.

s.5.1.5. The fraction of time the system contains n jobs is $\pi(n)$ (by PASTA). When the system contains $n>0$ jobs, the number in queue is one less, i.e., $n-1$.

$$\mathsf{E}[Q] = \sum_{n=1}^{\infty}(n-1)\pi(n) = \sum_{n=1}^{\infty} n\pi(n) - \sum_{n=1}^{\infty}\pi(n) = \mathsf{E}[L] - (1-\pi(0)) = \mathsf{E}[L] - \rho.$$

s.5.1.6. Take $\lambda(n) = (N-n)\lambda$ and $\mu(n) = n\mu$. Then

$$\begin{aligned} p(n+1) &= \frac{\lambda(n)}{\mu(n+1)}p(n) = \frac{(N-n)\lambda}{(n+1)\mu}p(n) = \frac{(N-n)(N-(n-1))}{(n+1)n}\frac{\lambda^2}{\mu^2}p(n-1) \\ &= \frac{N!}{(N-(n+1))!}\frac{1}{(n+1)!}\rho^{n+1}p(0) = \binom{N}{n+1}\rho^{n+1}p(0). \end{aligned}$$

Therefore, $G = \sum_{k=0}^{N}\rho^k\binom{N}{k}$.

s.5.1.7. Using the hint,

$$M_L(s) = \mathsf{E}\left[e^{sL}\right] = \sum_{n=0}^{\infty} e^{sn}p(n) = (1-\rho)\sum_{n} e^{sn}\rho^n = \frac{1-\rho}{1-e^s\rho},$$

where we assume that s is such that $e^s\rho < 1$. Then,

$$\begin{aligned} M_L'(s) = (1-\rho)\frac{1}{(1-e^s\rho)^2}e^s\rho \implies \mathsf{E}[L] = M_L'(0) &= \frac{\rho}{1-\rho}, \\ \mathsf{E}\left[L^2\right] = M''(0) &= \frac{2\rho^2}{(1-\rho)^2} + \frac{\rho}{1-\rho} = \rho\frac{1+\rho}{(1-\rho)^2}, \\ \mathsf{V}[L] = \mathsf{E}\left[L^2\right] - (\mathsf{E}[L])^2 &= \frac{\rho(1+\rho)}{(1-\rho)^2} - \frac{\rho^2}{(1-\rho)^2} = \frac{\rho}{(1-\rho)^2}, \\ \mathsf{P}\{L\geq n\} = \sum_{k=n}^{\infty}p(k) = \sum_{k=n}^{\infty}p(0)\rho^k = (1-\rho)\sum_{k=n}^{\infty}\rho^k &= (1-\rho)\rho^n\sum_{k=0}^{\infty}\rho^k = (1-\rho)\rho^n\frac{1}{1-\rho} = \rho^n. \end{aligned}$$

s.5.1.8. For $n\geq 1$,

$$\begin{aligned} p(n) &= \frac{\lambda(n-1)}{\mu(n)}p(n-1) = \frac{\lambda}{\min\{c,n\}\mu}p(n-1) = \frac{1}{\min\{c,n\}}(c\rho)p(n-1) \\ &= \frac{1}{\min\{c,n\}\min\{c,n-1\}}(c\rho)^2p(n-2) \\ &= \frac{1}{\prod_{k=1}^{n}\min\{c,k\}}(c\rho)^n p(0). \end{aligned}$$

To obtain the normalization constant $G = 1/p(0)$,

$$\begin{aligned}
1 &= \sum_{n=0}^{\infty} p(n) = p(0) + \sum_{n=1}^{c-1} p(n) + \sum_{n=c}^{\infty} p(n) \\
&= p(0) + p(0) \sum_{n=1}^{c-1} \frac{(c\rho)^n}{n!} + p(0) \sum_{n=c}^{\infty} \frac{c^c}{c!} \rho^n \\
&= p(0) \sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + p(0) \sum_{n=c}^{\infty} \frac{(c\rho)^c}{c!} \rho^{n-c} \\
&= p(0) \sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + p(0) \frac{(c\rho)^c}{c!} \sum_{n=0}^{\infty} \rho^n \\
&= p(0) \sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + p(0) \frac{(c\rho)^c}{c!(1-\rho)}.
\end{aligned}$$

Next, for $n \geq c$, note that $p(n) = (c\rho)^n/(c^{n-c}c!) = c^c\rho^n/c!$, so that

$$\begin{aligned}
\mathsf{E}[Q] &= \sum_{n=c}^{\infty} (n-c)p(n) = \sum_{n=c}^{\infty} (n-c) \frac{c^c}{c!} \rho^n p(0) \\
&= \frac{c^c\rho^c}{Gc!} \sum_{n=c}^{\infty} (n-c)\rho^{n-c} = \frac{c^c\rho^c}{Gc!} \sum_{n=0}^{\infty} n\rho^n \\
&= \frac{c^c\rho^c}{Gc!} \frac{\rho}{(1-\rho)^2}.
\end{aligned}$$

The derivation of the expected number of jobs in service becomes easier if we pre-multiply the normalization constant G:

$$\begin{aligned}
G\mathsf{E}[L_s] &= G\left(\sum_{n=0}^{c} np(n) + \sum_{n=c+1}^{\infty} cp(n) \right) \\
&= \sum_{n=1}^{c} n \frac{(c\rho)^n}{n!} + \sum_{n=c+1}^{\infty} c \frac{c^c\rho^n}{c!} = \sum_{n=1}^{c} \frac{(c\rho)^n}{(n-1)!} + \frac{c^{c+1}}{c!} \sum_{n=c+1}^{\infty} \rho^n \\
&= \sum_{n=0}^{c-1} \frac{(c\rho)^{n+1}}{n!} + \frac{(c\rho)^{c+1}}{c!} \sum_{n=0}^{\infty} \rho^n = c\rho \left(\sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!(1-\rho)} \right).
\end{aligned}$$

Observe that the RHS is precisely equal to ρcG, so that $\mathsf{E}[L_s] = c\rho$.

s.5.1.9. Take $c = 1$

$$\begin{aligned}
p(0) &= \frac{1}{G} \frac{(c\rho)^0}{0!} = \frac{1}{G}, \\
p(n) &= \frac{1}{G} \frac{c^c\rho^n}{c!} = \frac{1}{G} \frac{1^1\rho^n}{1!} = \frac{\rho^n}{G}, \quad n = 1,2,\ldots \\
G &= \sum_{n=0}^{c-1} \frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{(1-\rho)c!} = \sum_{n=0}^{0} \frac{\rho^0}{0!} + \frac{\rho}{(1-\rho)} = 1 + \frac{\rho}{1-\rho} = \frac{1}{1-\rho}, \\
\mathsf{E}[L] &= \frac{(c\rho)^c}{c!G} \frac{\rho}{(1-\rho)^2} = \frac{\rho}{1/(1-\rho)} \frac{\rho}{(1-\rho)^2} = \frac{\rho^2}{1-\rho}, \\
\mathsf{E}[L] &= \sum_{n=0}^{c} np(n) + \sum_{n=c+1}^{\infty} cp(n) = p(1) + 1 \sum_{n=2}^{\infty} p(n) = 1 - p(0) = \rho.
\end{aligned}$$

Everything is in accordance to the formulas we derived earlier for the $M/M/1$ queue.

s.5.1.10. With the other results for the $M/M/1/K$ queue,

$$1-p(0) = 1-\frac{1-\rho}{1-\rho^{K+1}} = \frac{1-\rho^{K+1}-1+\rho}{1-\rho^{K+1}} = \rho\frac{1-\rho^K}{1-\rho^{K+1}},$$
$$\rho(1-p(K)) = \rho\frac{1-\rho^{K+1}-\rho^K+\rho^{K+1}}{1-\rho^{K+1}} = \rho\frac{1-\rho^K}{1-\rho^{K+1}}.$$

s.5.1.11. To take the limit $K\to\infty$—mind, not the limit $n\to\infty$—, write

$$G = \frac{1-\rho^{K+1}}{1-\rho} = \frac{1}{1-\rho} - \frac{\rho^{K+1}}{1-\rho}.$$

Since $\rho^{K+1}\to 0$ as $K\to\infty$ (recall, $\rho<1$), we get

$$G\to\frac{1}{1-\rho},$$

as $K\to\infty$. Therefore, $p(n)=\rho^n/G\to\rho^n(1-\rho)$, and the latter are the steady-state probabilities of the $M/M/1$ queue. Finally, if the steady-state probabilities are the same, the performance measures (which are derived from $p(n)$) must be the same.

s.5.2.1. True.

s.5.2.2. False. This depends on the objective. The service time at a slow server is longer. However, when multiple servers are present, if one server breaks down, service can still continue with the other servers, while when a single fast server fails, everything stops.

s.5.2.3. True.

s.5.2.4. False. Just after jobs balk, the number of jobs in the system is less than K.

s.5.2.5. As people arrive without making appointments, it is reasonable to say that the inter-arrival times are memoryless. As for the services, the employee deals with many different questions, some are short, some are long. So, we model service times as exponential too.

With Little's law:

```
>>> labda = 5. # per minute
>>> EW = 6.
>>> EQ = labda * EW
>>> EQ
30.0
```

We can use the expression for $\mathsf{E}[W]$ to solve for ρ, but we can also do a simple search.

```
>>> from scipy.optimize import bisect

>>> EW = 6.

>>> def find_W(rho):
...     # return W -1 for given rho
...     ES = rho / labda
```

```
...     return rho / (1 - rho) * ES - EW
...

>>> rho = bisect(find_W, 0, 0.999)
>>> rho
0.968719422671953
>>> ES = rho / labda
>>> ES
0.1937438845343906
>>> J = EW + ES
>>> J
6.19374388453439
>>> L = labda*J
>>> L
30.968719422671953
```

Next, find n such that $\sum_{j=1}^{n} p_j > 0.9$. We start counting at 1, because while the system contains one job, this customer is in service, hence stands at desk of the employee.

```
>>> n, p = 0, 1 - rho
>>> total = p
>>> while total <= 0.9:
...     p *= rho
...     total += p
...     n += 1
...
>>> total
0.9017224011105635
>>> n - 1
71
```

Note that the while loop breaks when n is one too large.

The average queue is huge: hire an extra employee.

s.5.2.6. This is an $M/M/1/3$ queue: there is room for 1 customer in service and two in queue.

We import `numpy` and convert the lists to arrays is to fix the output precision to 3, otherwise we get long floats in the output.

First the case with $b = 2$.

```
>>> import numpy as np
>>> np.set_printoptions(precision=3)

>>> labda, mu, c = 6.0,  5.0, 1
>>> rho = labda / mu
>>> K = c + 2


>>> p = np.array([rho ** n for n in range(K + 1)]) # range(n) is up to n
>>> G = sum(p)
>>> p /= G  # normalize
>>> p
array([0.186, 0.224, 0.268, 0.322])
>>> L = sum(n * p[n] for n in range(len(p)))
```

```
>>> L
1.725782414307004
>>> Q = sum(max(n - c,0) * p[n] for n in range(len(p)))
>>> Q
0.9120715350223545
>>> lost = labda * p[-1]  # the last element of p
>>> labda - lost # accepted, hence served
4.06855439642325
```

Now with three extra chairs.

```
>>> K += 3
>>> K
6
>>> p = np.array([rho ** n for n in range(K + 1)]) # range(n) is up to n
>>> G = sum(p)
>>> p /= G  # normalize
>>> lost = labda * p[-1]  # the last element of P
>>> labda - lost # accepted, hence served
4.612880368265357
```

We see that since the server is overloaded, the acceptance is not much affected by increasing the number of chairs. We need an extra server.

s.5.2.7. The computations.

```
>>> import numpy as np
>>> np.set_printoptions(precision=3)

>>> labda = np.array([10.0, 10.0, 20.0, 20.0, 5.0])
>>> mu = np.array([0., 30., 60., 60., 60., 60.])
>>> c = 2

>>> p = np.ones_like(mu)
>>> for i in range(len(labda)):
...     p[i+1] = labda[i] *p[i]/ mu[i+1]
...
>>> p /= p.sum()
>>> p
array([7.072e-01, 2.357e-01, 3.929e-02, 1.310e-02, 4.365e-03, 3.638e-04])
>>> labdaBar = sum(labda[n] * p[n] for n in range(len(labda)))
>>> labdaBar
10.498363041105856
>>> L = sum(n * p[n] for n in range(len(p)))
>>> L
0.37286285922153506
>>> Q = sum(max(n - c,0) * p[n] for n in range(len(p)))
>>> Q
0.022917424518006546
>>> J = L / labdaBar
>>> J
0.035516285516285516
>>> W = Q / labdaBar
```

```
>>> W
0.002182952182952183
```

s.5.2.8. From the hint, the rv L behaves like the number of jobs in an $M/M/1$ queue. If we have L, then with the two properties $TG = 0$ and $3 = T + L - G$, we can retrieve (uniquely at all moments in time) the number of pattys T on stock and the number of people waiting G. In inventory terms, G represents the number of *backlogged* pattys.

With this mapping, the expected number of pattys on stock is $\mathsf{E}[T] = \sum_{l=0}^{3}(3-l)p(l)$.

```
>>> labda = 12.0  # per hour
>>> mu = 15.0  # per hour
>>> rho = labda / mu
>>> max_pattys = 3

>>> ET, p = 0, 1 - rho
>>> for l in range(max_pattys):
...     ET += (max_pattys - l) * p
...     p *= rho
...
>>> ET
1.0479999999999998
```

For the expected number of people waiting, use that $3 = T - L + G$ always. Taking expectations, we see that $3 = \mathsf{E}[T] + \mathsf{E}[L] - \mathsf{E}[G]$, where $\mathsf{E}[G]$ is the expected number of people.

```
>>> EL = rho/(1-rho)
>>> EG = ET + EL - max_pattys
>>> EG
2.048000000000001
```

Computing the waiting times is tricky. For the pattys, the rate at which 'jobs' arrive, is the arrival rate of people. For the people, the rate at which 'jobs' arrive is the arrival rate of pattys.

```
>>> ET/labda # Waiting for pattys
0.08733333333333332
>>> EG/mu # waiting time for people
0.1365333333333334
```

What would be the impact of allowing 4 pattys maximally?

s.5.2.9. The *invariant* $3 = T + L - G$ becomes $4 = T + L - G$.

Estimating the impact of allowing maximally four pattys is simple with the code we already have.

```
>>> max_pattys = 4

>>> ET, p = 0, 1 - rho
>>> for l in range(max_pattys):
...     ET += (max_pattys - l) * p
...     p *= rho
```

```
...
>>> ET
1.6383999999999999
>>> EG = ET + EL - max_pattys
>>> EG
1.6384000000000007
```

s.5.2.10. The *invariant* $3 = T + L - G$ becomes $0 = T + L - G$. This implies that $T = 0$ and $L = G$. Hence $\mathsf{E}[G] = \mathsf{E}[L]$.

s.5.3.1. True.

s.5.3.2. True. $\mathsf{V}[B] = 0 \implies C_B^2 = 0 \implies$ the first term on the RHS does not change when ρ remains constant. The second term does become smaller.

s.5.3.3. False. Jobs come in batches, hence, multiple items can arrive at the same time, implying that the interarrival distribution is no longer exponential.

s.5.3.4. False. The range over m should include n; now it runs up to (but not including) n.

s.5.3.5. Start with the simple case. $B \equiv 2 \implies \mathsf{V}[B] = 0 \implies C_s^2 = 0$, $\rho = \lambda\,\mathsf{E}[B]\,\mathsf{E}[S] = 1{\cdot}2{\cdot}25/60 = 5/6$. Hence,

$$\mathsf{E}[L] = \frac{1}{2}\frac{5/6}{1/6}2 + \frac{1}{2}\frac{5/6}{1/6} = 5 + \frac{5}{2}.$$

Now the other case. $\mathsf{E}\left[B^2\right] = (1+4+9)/3 = 14/3$. Hence, $\mathsf{V}[B] = 14/3 - 4 = 2/3$. Hence, $C_s^2 = \frac{1}{6}$. And thus,

$$\mathsf{E}[L] = \frac{1+1/6}{2}\frac{5/6}{1/6}2 + \frac{1}{2}\frac{5/6}{1/6} = \frac{7}{6}5 + \frac{5}{2}.$$

The ratio between $\mathsf{E}[L]$ is 10/9: the average waiting time can be reduced by about 10% by working in fixed batch sizes.

s.5.3.6. Use the hint. For a geometric rv,

$$\mathsf{E}[B] = \frac{1}{p}, \qquad \mathsf{V}[B] = \frac{1}{p^2} - \frac{1}{p}, \qquad C_B^2 = \frac{\mathsf{V}[B]}{(\mathsf{E}[B])^2} = p^2\left(\frac{1}{p^2} - \frac{1}{p}\right) = 1 - p,$$
$$(1 + C_B^2)/2 = 1 - p/2,$$

With this, fill in the formula for $\mathsf{E}[W]$ to get

$$\mathsf{E}[L] = \left(1 - \frac{p}{2}\right)\frac{\rho}{1-\rho}\frac{1}{p} + \frac{1}{2}\frac{\rho}{1-\rho} = \frac{\rho}{1-\rho}\frac{1}{p}.$$

For the $M/M/1$ queue, $\mathsf{E}[B] = 1 \implies p = 1 \implies \mathsf{E}[L] = \rho/(1-\rho)$.

If you can't find the expectation and variance of a geometric rv, then here is the derivation.

$$\begin{aligned}
M_B(s) &= \mathsf{E}\left[e^{sB}\right] = \sum_{k=0}^{\infty} e^{sk}\,\mathsf{P}\{B=k\} \\
&= \sum_{k=0}^{\infty} e^{sk} p q^{k-1} = \frac{p}{q}\sum_{k=0}^{\infty}(qe^s)^k = \frac{p}{q}\frac{1}{1-qe^s}, \\
\mathsf{E}[B] = M_B'(0) &= \frac{p}{q}\frac{q}{(1-qe^s)^2}\bigg|_{s=0} = \frac{p}{(1-q)^2} = \frac{1}{p}, \\
\mathsf{E}\left[B^2\right] = M_B''(0) &= \frac{2}{p^2} - \frac{1}{p}, \\
\mathsf{V}[B] &= \mathsf{E}\left[B^2\right] - (\mathsf{E}[B])^2 = \frac{2}{p^2} - \frac{1}{p} - \frac{1}{p^2} = \frac{1}{p^2} - \frac{1}{p},
\end{aligned}$$

s.5.3.7. For the partial acceptance case, any job is accepted, but the system only admits whatever fits. As level $n \in 0,1,\ldots,K-1$ is still up-crossed by any batch of size at least $n-m$ when the system is in state m, the formula for the up-crossing rate is identical to the case without this acceptance policy. Hence, $\mu\pi(n+1) = \lambda\sum_{m=0}^{n}\pi(m)G(n-m)$, for $n = 0,1,\ldots,K-1$.

This is the code.

```
import numpy as np

from random_variable import RV

labda, mu = 1, 3
rho = labda / mu
f = {1: 1, 2: 1, 3: 1}
S = RV(f)


def partial_acceptance(K):
    pi, n = {}, 0
    pi[0] = 1
    while n < K:
        pi[n + 1] = sum(pi[m] * S.sf(n - m) for m in range(n + 1))
        pi[n + 1] *= rho
        n += 1
    return RV(pi)


pi = partial_acceptance(5)
print(pi.mean())
```

s.5.3.8. The complete-acceptance policy is actually quite simple. As any batch will be accepted when $n \le K$, the queue length is not bounded. Only when the number of items in the system is larger than K, we do not accept jobs.

$$\mu\pi(n+1) = \begin{cases} \lambda\sum_{m=0}^{n}\pi(m)G(n-m), & \text{for } n \le K, \\ \lambda\sum_{m=0}^{K}\pi(m)G(n-m), & \text{for } n > K. \end{cases}$$

Use the code of [5.3.7] to get a working example.

```
def complete_acceptance(K):
    pi, n = {}, 0
    pi[0] = 1
    while S.sf(n - K) > 0:
        pi[n + 1] = sum(pi[m] * S.sf(n - m) for m in range(min(n, K) + 1))
        pi[n + 1] *= rho
        n += 1
    return RV(pi)
```

s.5.3.9. Suppose a batch of size k arrives when the system contains m items. When $m+k \le K$, the batch can be accepted since the entire batch will fit into the queue, otherwise it will be rejected. Further, level $n, 0 \le n < K$, can only be crossed when $m+k > n$. Thus, for $n = 0, \ldots, K-1$,

$$\begin{aligned}\mu\pi(n+1) &= \lambda \sum_{m=0}^{n} \pi(m)\,\mathsf{P}\{n-m < B \le K-m\} \\ &= \lambda \sum_{m=0}^{n} \pi(m)[G(n-m) - G(K-m)],\end{aligned}$$

Let us check this for some simple cases. First, when $K \to \infty$, then $G(K-m) \to 0$, so we get our earlier result. Second, take $n = K$. Then $G(n-m) - G(K-m) = 0$ for all m, so that the RHS is 0, as it should. Third, take $n = 0$ and $K = 1$, then $\mu\pi(1) = \lambda\pi(0)$. This also makes sense.

Use the code of [5.3.7] to get a working example.

```
def complete_rejection(K):
    pi, n = {}, 0
    pi[0] = 1
    while n < K:
        pi[n + 1] = sum(
            pi[m] * (S.sf(n - m) - S.sf(K - m)) for m in range(n + 1)
        )
        pi[n + 1] *= rho
        n += 1
    return RV(pi)
```

s.5.3.10. For 1, observe first that $\sum_{k=0}^{\infty} \mathbb{1}_{m>k} = m$, since $\mathbb{1}_{m>k} = 1$ if $k < m$ and $\mathbb{1}_{m>k} = 0$ if $k \ge m$. With this,

$$\begin{aligned}\sum_{k=0}^{\infty} G(k) &= \sum_{k=0}^{\infty} \mathsf{P}\{X > k\} = \sum_{k=0}^{\infty} \sum_{m=k+1}^{\infty} \mathsf{P}\{X = m\} \\ &= \sum_{k=0}^{\infty} \sum_{m=0}^{\infty} \mathbb{1}_{m>k}\,\mathsf{P}\{X = m\} = \sum_{m=0}^{\infty} \sum_{k=0}^{\infty} \mathbb{1}_{m>k}\,\mathsf{P}\{X = m\} \\ &= \sum_{m=0}^{\infty} m\,\mathsf{P}\{X = m\} = \mathsf{E}[X].\end{aligned}$$

There are some technical details with respect to the interchange of the summations. Since the summands are positive, this is allowed.

For 2,

$$\begin{aligned}\sum_{i=0}^{\infty} iG(i) &= \sum_{i=0}^{\infty} i \sum_{n=i+1}^{\infty} \mathsf{P}\{X=n\} = \sum_{n=0}^{\infty} \mathsf{P}\{X=n\} \sum_{i=0}^{\infty} i\, \mathbb{1}_{n\geq i+1} \\ &= \sum_{n=0}^{\infty} \mathsf{P}\{X=n\} \sum_{i=0}^{n-1} i = \sum_{n=0}^{\infty} \mathsf{P}\{X=n\} \frac{(n-1)n}{2} \\ &= \sum_{n=0}^{\infty} \frac{n^2}{2} \mathsf{P}\{X=n\} - \frac{\mathsf{E}[X]}{2} = \frac{\mathsf{E}\left[X^2\right]}{2} - \frac{\mathsf{E}[X]}{2}.\end{aligned}$$

s.5.3.11.

$$\sum_{i=1}^{\infty} iG(i-1) = \sum_{i=0}^{\infty} (i+1)G(i) = \sum_{i=0}^{\infty} iG(i) + \sum_{i=0}^{\infty} G(i) = (\mathsf{E}\left[B^2\right] - \mathsf{E}[B])/2 + \mathsf{E}[B].$$

s.5.3.12. We use that $\mu\pi(n) = \lambda \sum_{i=0}^{n-1} \pi(i)G(n-1-i)$ and the results of the exercises of Section 5.3 to see that

$$\begin{aligned}\mu\,\mathsf{E}[L] &= \sum_{n=0}^{\infty} n\,\mu\pi(n), \quad \text{now substitute for } \mu\pi(n) \text{ the recursion (5.3.4)}, \\ &= \lambda \sum_{n=0}^{\infty} n \sum_{i=0}^{n-1} \pi(i)G(n-1-i) = \lambda \sum_{n=0}^{\infty} n \sum_{i=0}^{\infty} \mathbb{1}_{i<n}\pi(i)G(n-1-i) \\ &= \lambda \sum_{i=0}^{\infty} \pi(i) \sum_{n=0}^{\infty} \mathbb{1}_{i<n} nG(n-1-i) = \lambda \sum_{i=0}^{\infty} \pi(i) \sum_{n=i+1}^{\infty} nG(n-1-i) \\ &= \lambda \sum_{i=0}^{\infty} \pi(i) \sum_{n=0}^{\infty} (n+i+1)G(n) = \lambda \sum_{i=0}^{\infty} \pi(i) \left[\sum_{n=0}^{\infty} nG(n) + (i+1) \sum_{n=0}^{\infty} G(n) \right] \\ &= \lambda \sum_{i=0}^{\infty} \pi(i) \sum_{n=0}^{\infty} nG(n) + \lambda\,\mathsf{E}[B] \sum_{i=0}^{\infty} \pi(i)(i+1) \\ &= \lambda \sum_{i=0}^{\infty} \pi(i) \frac{\mathsf{E}\left[B^2\right] - \mathsf{E}[B]}{2} + \lambda\,\mathsf{E}[B]\,(\mathsf{E}[L]+1) \\ &= \lambda \frac{\mathsf{E}\left[B^2\right] - \mathsf{E}[B]}{2} + \lambda\,\mathsf{E}[B]\,\mathsf{E}[L] + \lambda\,\mathsf{E}[B] \\ &= \lambda \frac{\mathsf{E}\left[B^2\right]}{2} + \lambda\,\mathsf{E}[B]\,\mathsf{E}[L] + \lambda \frac{\mathsf{E}[B]}{2}.\end{aligned}$$

Dividing both sides by μ and using that $\lambda\,\mathsf{E}[B]/\mu = \rho$, we obtain

$$EL = \lambda \frac{\mathsf{E}\left[B^2\right]}{2} \mathsf{E}[S] + \rho\,\mathsf{E}[L] + \frac{\rho}{2}.$$

By bringing $\rho\,\mathsf{E}[L]$ to the LHS, the RHS becomes equal to (5.3.2).

s.5.4.1. True.

s.5.4.2. True.

s.5.4.3. False. It should be this: $d_k = I_k \min\{L_{k-1}, c_k\}$, $L_k = L_{k-1} - d_k + a_k$.

s.5.4.4. True.

s.5.4.5. When we switch on the server, the queue 'drains' at rate $\mu - \lambda > 0$, with $\mu = 1/\mathsf{E}[S]$. Consequently, no matter how large N, $T(N) < \infty$. And, whenever the system is empty, the stochastic process restarts. As such cycles start over and over again, and the queue length can never 'escape to infinity'.

s.5.4.6. The total number $A(t)$ of job that arrive during $[0,t]$ does not depend on N. Thus, in (4.3.2), $\sum_{k=1}^{A(t)} S_k$ does not depend on N. Now use rate-stability.

s.5.4.7. $\rho = \lambda\,\mathsf{E}[S] = (3/8)\cdot 2 = 3/4$, $\mathsf{E}[W] = 4.5$ h. If we were able to reduce all service variability, i.e., $C_s^2 = 0$, then still $\mathsf{E}[W] = 3$h. Hence, we have to increase capacity, or reduce $\mathsf{E}[S]$. Another possibility is to plan the arrival of jobs such that $C_a^2 = 0$. However, typically this is not possible. Would you accept that the supermarket plans your visits?

s.5.4.8. $\mathsf{V}[S] = 0 \implies C_s^2 = 0 \implies \mathsf{E}[W_{M/D/1}] = \mathsf{E}[W_{M/M/1}]/2$.

s.5.4.9.

$$\mathsf{E}[S] = \alpha/2, \qquad \mathsf{E}\left[S^2\right] = \int_0^\alpha x^2\,\mathrm{d}x/\alpha = \alpha^2/3,$$
$$\mathsf{V}[S] = \alpha^2/3 - \alpha^2/4 = \alpha^2/12, \qquad C_s^2 = (\alpha^2/12)/(\alpha^2/4) = 1/3,$$
$$\rho = \lambda\alpha/2,$$
$$\mathsf{E}[W] = \frac{1+C_s^2}{2}\frac{\lambda\alpha/2}{1-\lambda\alpha/2}\frac{\alpha}{2}, \qquad \mathsf{E}[J] = \mathsf{E}[W] + \frac{\alpha}{2}.$$

s.5.4.10. The probability to find the server busy upon arrival is ρ, and only jobs that find the server occupied see a positive remaining service time $S_r > 0$.

$$\mathsf{E}[S_r] = \rho\,\mathsf{E}[S_r \mid S_r > 0] + (1-\rho)\,\mathsf{E}[S_r \mid S_r = 0] = \rho\,\mathsf{E}[S_r \mid S_r > 0]\,.$$

For the $M/M/1$, service times are memoryless, hence, $\mathsf{E}[S_r \mid S_r > 0] = \mathsf{E}[S]$, and this implies that $\mathsf{E}[S_r] = \rho\,\mathsf{E}[S]$ for the $M/M/1$ queue.

s.5.4.11. The cost up to the qth job is the cost $W(q-1)$ up to the arrival of job $q-1$ plus the cost while there are $q-1$ jobs in the system. The time between the arrival of job $q-1$ and q is $1/\lambda$. When there are $q-1$ jobs in the system, the expected cost is therefore $h(q-1)/\lambda$ during the arrival of job $q-1$ and job q. And thus, $W(q) = h\sum_{i=1}^{q}(i-1)$.

s.5.4.12. Consider an arbitrary moment in time at which $q > 0$ and the server is busy. Now either of two events happens first: a new job enters the system, or the job in service leaves. The probability of an arrival to occur first is $\alpha = \lambda/(\lambda+\mu)$, the probability of a departure to occur first is $\beta = 1-\alpha = \mu/(\lambda+\mu)$. Moreover, the expected time to either an arrival or a departure, whichever is first, is $1/(\mu+\lambda)$. In words, the system stays in state q for an expected time $1/(\lambda+\mu)$ until an arrival or departure occurs. Then, it moves to state $q+1$ or $q-1$, and from there it takes $T(q+1)$ or $T(q-1)$ until the system is empty. Observe that this reasoning depends crucially on the memoryless property.

s.5.4.13. Note that the queueing cost is hq per unit time when there are q jobs in the system, it costs $hq/(\lambda+\mu)$ until an arrival or departure occurs. For the rest we follow the reasoning by which we derive the recursion for $T(q)$ for the $M/M/1$ queue.

s.5.4.14. In the hint, the first equation is superfluous. In the second, bq cancels at both sides, by which we find a. The third now follows.

s.5.4.15. For b, using the expressions for $\mathsf{E}[Y]$ and $\mathsf{E}\left[Y^2\right]$,

$$\begin{aligned}
b(1-\mathsf{E}[Y]) &= a(\mathsf{E}\left[Y^2\right]-2\mathsf{E}[Y]+1)+\frac{1}{2}h\lambda\mathsf{E}\left[S^2\right] \\
&= \frac{h\mathsf{E}[S]}{2(1-\mathsf{E}[Y])}(\mathsf{E}\left[Y^2\right]-2\mathsf{E}[Y]+1)+\frac{1}{2}h\lambda\mathsf{E}\left[S^2\right] \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(\lambda^2\mathsf{E}\left[S^2\right]+\lambda\mathsf{E}[S]-2\lambda\mathsf{E}[S]+1+\lambda\mathsf{E}\left[S^2\right]\frac{1-\lambda\mathsf{E}[S]}{\mathsf{E}[S]}\right) \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(\lambda^2\mathsf{E}\left[S^2\right]-\lambda\mathsf{E}[S]+1+\frac{\lambda\mathsf{E}\left[S^2\right]}{\mathsf{E}[S]}-\lambda^2\mathsf{E}\left[S^2\right]\right) \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(1+\frac{\lambda\mathsf{E}\left[S^2\right]}{\mathsf{E}[S]}-\lambda\mathsf{E}[S]\right) \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(1+\frac{\lambda\mathsf{E}\left[S^2\right]}{\mathsf{E}[S]}-\lambda\frac{(\mathsf{E}[S])^2}{\mathsf{E}[S]}\right) \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(1+\lambda\frac{\mathsf{V}[S]}{\mathsf{E}[S]}\right) \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(1+\lambda\frac{\mathsf{V}[S]}{(\mathsf{E}[S])^2}\mathsf{E}[S]\right) \\
&= \frac{h\mathsf{E}[S]}{2(1-\lambda\mathsf{E}[S])}\left(1+\rho C_s^2\right).
\end{aligned}$$

Divide now both sides by $1-\mathsf{E}[Y]$.

s.5.4.16. Note first that $C(N)=N(1/\lambda+\mathsf{E}[S]/(1-\rho))=N/(\lambda(1-\rho))$. Then,

$$\begin{aligned}
\frac{V(N)+K+W(N)}{C(N)} &= \left(aN^2+bN+K+hN(N-1)/2\lambda\right)\frac{\lambda(1-\rho)}{N} \\
&= \frac{h}{2}\rho N+\frac{h}{2}\frac{\rho}{1-\rho}(1+\rho C_s^2)+\frac{h}{2}(N-1)(1-\rho)+K\frac{\lambda(1-\rho)}{N} \\
&= \frac{h}{2}\frac{\rho}{1-\rho}(1+\rho C_s^2)+\frac{h}{2}(N-1+\rho)+K\frac{\lambda(1-\rho)}{N} \\
&= \frac{h}{2}\frac{\rho}{1-\rho}(\rho+\rho C_s^2+1-\rho)+\frac{h}{2}(N-1+\rho)+K\frac{\lambda(1-\rho)}{N} \\
&= \frac{h}{2}\frac{\rho^2}{1-\rho}(1+C_s^2)+\frac{h}{2}\rho+\frac{h}{2}(N-1+\rho)+K\frac{\lambda(1-\rho)}{N} \\
&= \frac{h}{2}\frac{\rho^2}{1-\rho}(1+C_s^2)+h\rho+h\frac{N-1}{2}+K\frac{\lambda(1-\rho)}{N}.
\end{aligned}$$

s.5.5.1. True. This production inventory system is equivalent to an $M/M/1$ queue.

s.5.5.2. In continuous time, when $I_k = 0$, the kth demand 'sees' an empty inventory. Hence, the kth demand cannot be met from on-hand stock when $I_k = 0$. Thus, only when $I_k > 0$, this demand can be served. Therefore,

$$\begin{aligned}\alpha(s) &= \mathsf{P}\{I_k > 0\}\\ &= \mathsf{P}\{s+1-D[k-L,k] > 0\}\\ &= \mathsf{P}\{D[k-L,k] < s+1\}\\ &= \mathsf{P}\{D[k-L,k] \leq s\}\\ &= F(S).\end{aligned}$$

s.6.1.1. True.

s.6.1.2. True. Suppose for ease that the SCVs don't change when we reduce the load ρ by 5%, and that we achieve this load by reducing λ with 5% so that $\mathsf{E}[S]$ remains the same too. Then the only factor that changes in Sakasegawa's formula is $\rho/(1-\rho)$. Now, when $\rho = 0.95$, this factor is $0.95/(1-0.95) \approx 20$. When ρ becomes 0.9, then $\rho/(1-\rho) = 0.9/(1-0.9) \approx 10$. Since 10 is half of 20, the reduction in waiting time is roughly a factor 2.

s.6.1.3. True. If the queue is not stable, $\mathsf{E}[W] = \infty$.

s.6.1.4. When processing times at a station are nearly constant, and the jobs of this station are sent to a second station for further processing, the inter-arrival times at the second station must be roughly equal. But then the inter-arrival times are not well approximated by the exponential distribution, consequently, the arrival process is not well described by a Poisson process.

s.6.1.5. $\rho = \lambda\,\mathsf{E}[S] = 1/60 \cdot 50 = 5/6$. Since job arrivals do not overlap any job service, the number of jobs in the system is 1 for 50 seconds, then the server is idle for 10 seconds, and so on. Thus $\mathsf{E}[L] = 1 \cdot 5/6 = 5/6$. There is no variance in the inter-arrival times, and also not in the service times, thus $C_a^2 = C_s^2 = 0$. Also $\mathsf{E}[W] = 0$ since $\mathsf{E}[Q] = 0$.

s.6.1.6. Again $\mathsf{E}[S]$ is 50 seconds, so that $\rho = 5/6$. Also $C_a^2 = 0$. For the C_s^2 we have to do some work.

$$\begin{aligned}\mathsf{E}[S] &= \frac{20}{2} + \frac{80}{2} = 50\\ \mathsf{E}\left[S^2\right] &= \frac{400}{2} + \frac{6400}{2} = 3400\\ \mathsf{V}[S] &= \mathsf{E}\left[S^2\right] - (\mathsf{E}[S])^2 = 3400 - 2500 = 900\\ C_s^2 &= \frac{\mathsf{V}[S]}{(\mathsf{E}[S])^2} = \frac{900}{2500} = \frac{9}{25}.\end{aligned}$$

s.6.1.7. First the $G/G/1$ case. Observe that in this case, the inter-arrival time $X \sim U[3,9]$, that is, never smaller than 3 minutes, and never longer than 9 minutes.

```
>>> a = 3.0
>>> b = 9.0
>>> EX = (b + a) / 2.0  # expected inter-arrival time
>>> EX
```

```
6.0
>>> labda = 1.0 / EX  # per minute
>>> labda
0.16666666666666666
>>> VA = (b - a) * (b - a) / 12.0
>>> CA2 = VA / (EX * EX)
>>> CA2
0.08333333333333333

>>> ES = 5.0
>>> VS = 4
>>> CS2 = VS / (ES * ES)
>>> CS2
0.16

>>> rho = labda * ES
>>> rho
0.8333333333333333

>>> W = (CA2 + CS2) / 2.0 * rho / (1.0 - rho) * ES
>>> W
3.041666666666665
```

Now the $M/G/1$ case. In that case $C_a^2 = 1$.

```
>>> W = (1.0 + CS2) / 2.0 * rho / (1.0 - rho) * ES
>>> W
14.499999999999993
```

The arrival process with uniform inter-arrival times is much more regular than a Poisson process. In the first case, tool arrivals are spaced in time at least with 3 minutes.

s.6.1.8. With the hint,

$$\begin{aligned}
\mathsf{E}[S] &= \mathsf{E}[\mathbb{1}_{T=1} S_1] + \mathsf{E}[\mathbb{1}_{T=2} S_2] \\
&= \mathsf{E}[\mathbb{1}_{T=1}]\,\mathsf{E}[S_1] + \mathsf{E}[\mathbb{1}_{T=2}]\,\mathsf{E}[S_2], \quad \text{by the independence of } T, \\
&= \mathsf{P}\{T=1\}\,\mathsf{E}[S_1] + \mathsf{P}\{T=2\}\,\mathsf{E}[S_2] \\
&= p\,\mathsf{E}[S_1] + q\,\mathsf{E}[S_2].
\end{aligned}$$

For the variance, we need some algebra. Since,

$$\mathbb{1}_{T=1}\,\mathbb{1}_{T=2} = 0 \text{ and } \mathbb{1}_{T=1}^2 = \mathbb{1}_{T=1},$$

we get

$$\begin{aligned}
\mathsf{V}[S] &= \mathsf{E}\left[S^2\right] - (\mathsf{E}[S])^2 \\
&= \mathsf{E}\left[(\mathbb{1}_{T=1}S_1 + \mathbb{1}_{T=2}S_2)^2\right] - (\mathsf{E}[S])^2 \\
&= \mathsf{E}\left[\mathbb{1}_{T=1}S_1^2 + \mathbb{1}_{T=2}S_2^2\right] - (\mathsf{E}[S])^2 \\
&= p\,\mathsf{E}\left[S_1^2\right] + q\,\mathsf{E}\left[S_2^2\right] - (\mathsf{E}[S])^2 \\
&= p\,\mathsf{V}[S_1] + p(\mathsf{E}[S_1])^2 + q\,\mathsf{V}[S_2] + q(\mathsf{E}[S_2])^2 - (\mathsf{E}[S])^2 \\
&= p\,\mathsf{V}[S_1] + p(\mathsf{E}[S_1])^2 + q\,\mathsf{V}[S_2] + q(\mathsf{E}[S_2])^2 - p^2(\mathsf{E}[S_1])^2 - q^2(\mathsf{E}[S_2])^2 - 2pq\,\mathsf{E}[S_1]\,\mathsf{E}[S_2] \\
&= p\,\mathsf{V}[S_1] + q\,\mathsf{V}[S_2] + pq(\mathsf{E}[S_1])^2 + pq(\mathsf{E}[S_2])^2 - 2pq\,\mathsf{E}[S_1]\,\mathsf{E}[S_2], \quad \text{as } p = 1-q \\
&= p\,\mathsf{V}[S_1] + q\,\mathsf{V}[S_2] + pq(\mathsf{E}[S_1] - \mathsf{E}[S_2])^2.
\end{aligned}$$

s.6.2.1. True, [6.2.4]

s.6.2.2. False.

s.6.2.3. First check the load.

```
>>> labda = 3  # per hour
>>> ES0 = 15.0 / 60  # hour
>>> ES0
0.25
>>> ER = 2.0
>>> Bmin = labda * ER / (1 - labda * ES0)
>>> Bmin
24.0
```

```
>>> B = 30
>>> ES = ES0 + ER / B
>>> rho = labda * ES
>>> rho
0.95
```

The time to form a red batch is

```
>>> labda_r = 0.5
>>> EW_r = (B - 1) / (2 * labda_r)
>>> EW_r  # in hours
29.0
```

And the time to form a blue batch is

```
>>> labda_b = labda - labda_r
>>> EW_b = (B - 1) / (2 * labda_b)
>>> EW_b  # in hours
5.8
```

The time a batch spends in queue.

```
>>> Cae = 1.0
>>> CaB = Cae / B
>>> CaB
0.03333333333333333
>>> Ce = 1.0  # SCV of service times
>>> VS0 = Ce * ES0 * ES0
>>> VS0
0.0625
>>> VR = 1.0 * 1.0  # Var setups is sigma squared
>>> VS = B * VS0 + VR
>>> VS
2.875
>>> ESb = B * ES0 + ER
>>> ESb
9.5
>>> CeB = VS / (ESb * ESb)
>>> CeB
0.03185595567867036
>>> EW = (CaB + CeB) / 2 * rho / (1 - rho) * ESb
>>> EW
5.8833333333333275
```

The time to unpack the batch, i.e., the time at the server.

```
>>> Eunpack = ER + (B - 1) / 2 * ES0 + ES0
>>> Eunpack
5.875
```

The overall time red jobs spend in the system.

```
>>> total = EW_r + EW + Eunpack
>>> total
40.758333333333326
```

s.6.2.4. Suppose a batch is just finished. The first job of a new batch needs to wait, on average, $B-1$ inter-arrival times until the batch is complete, the second $B-2$ inter-arrival times, and so on. The last job does not have to wait at all. Thus, the total time to form a batch is $(B-1)/\lambda_r$. An arbitrary job can be anywhere in the batch, hence its expected time is half the total time.

s.6.2.5. The variance of the inter-arrival time of batches is B times the variance of job inter-arrival times. The inter-arrival times of batches is also B times the inter-arrival times of jobs. Thus,

$$C_{a,B}^2 = \frac{B\mathsf{V}[X]}{(B\mathsf{E}[X])^2} = \frac{\mathsf{V}[X]}{(\mathsf{E}[X])^2}\frac{1}{B} = \frac{C_a^2}{B}.$$

s.6.2.6. The variance of a batch is $\mathsf{V}\left[R+\sum_{i=1}^{B} S_{0,i}\right] = \mathsf{V}[R] + B\mathsf{V}[S_0]$, since the normal service times $S_{0,i}$, $i = 1,\ldots,B$, of the jobs are independent, and also independent of the setup time R of the batch.

s.6.2.7. First, wait until the setup is finished, then wait (on average) for half of the batch (minus the job itself) to be served, and then the job has to be served itself, that is, $\mathsf{E}[R] + \frac{B-1}{2}\mathsf{E}[S_0] + \mathsf{E}[S_0]$.

s.6.3.1. False.

s.6.3.2. False. Having fewer but larger breakdowns increases the variance more. This is bad from a queueing perspective.

s.6.3.3. First we determine the load.

```
>>> EB = 30
>>> p = 1 / EB
>>> ES0 = 1.5
>>> labda = 9.0 / (2 * 8)  # arrival rate per hour
>>> ER = 5.0
>>> ES = ES0 + p * ER
>>> ES
1.6666666666666667
>>> rho = labda * ES
>>> rho
0.9375
```

So, at least the system is stable.

```
>>> VS0 = 0.5 * 0.5
>>> VR = 2.0 * 2.0
>>> VS = VS0 + p * VR + p * (1 - p) * ER * ER
>>> VS
1.1888888888888887
>>> Ce2 = VS / (ES * ES)
>>> Ce2
0.4279999999999999
```

And now we can fill in the waiting time formula.

```
>>> Ca2 = 1  # Poisson arrivals
>>> EW = (Ca2 + Ce2) / 2 * rho / (1 - rho) * ES
>>> EW
17.849999999999998
>>> EJ = EW + ES
>>> EJ
19.516666666666666
```

s.6.3.4. We can use the model of Section 6.2.

```
>>> B = 20
>>> ER = 4.5
>>> VR = 0
>>> ES = ES0 + ER / B
>>> ES
1.725
>>> rho = labda * ES
>>> rho
0.9703125
```

```
>>> VS = VS0 + VR / B
>>> Ce2 = VS / (ES * ES)
>>> Ce2
0.08401596303297626
>>> EW = (Ca2 + Ce2) / 2 * rho / (1 - rho) * ES
>>> EW
30.55855263157897
>>> EJ = EW + ES
>>> EJ
32.28355263157897
```

Comparing this to the results of [6.3.3], we see that the load becomes somewhat higher. Since ρ becomes close to one, doing adjustments regularly is not a good idea.

s.6.3.5. Applying adam's and Eve's is straightforward, but the details require attention. Using independence where necessary (and allowed by our assumptions),

$$\begin{aligned}
\mathsf{E}[S|F] &= \mathsf{E}[S_0 + RF|F] = \mathsf{E}[S_0] + F\,\mathsf{E}[R]\,,\\
\mathsf{E}[S] &= \mathsf{E}[\mathsf{E}[S|F]] = \mathsf{E}[S_0] + \mathsf{E}[F]\,\mathsf{E}[R] = \mathsf{E}[S_0] + p\,\mathsf{E}[R]\,,\\
\mathsf{V}[\mathsf{E}[S|F]] &= \mathsf{V}[\mathsf{E}[S_0] + F\,\mathsf{E}[R]] = (\mathsf{E}[R])^2\,\mathsf{V}[F] = (\mathsf{E}[R])^2 p(1-p),\\
\mathsf{V}[S|F] &= \mathsf{V}[S_0 + RF|F] = \mathsf{V}[S_0] + F\,\mathsf{V}[R]\,,\\
\mathsf{E}[\mathsf{V}[S|F]] &= \mathsf{V}[S_0] + p\,\mathsf{V}[R]\,,\\
\mathsf{V}[S] &= \mathsf{E}[VS|F] + \mathsf{V}[\mathsf{E}[S|F]] = \mathsf{V}[S_0] + p\,\mathsf{V}[R] + p(1-p)(\mathsf{E}[R])^2.
\end{aligned}$$

s.6.4.1. False. The two expections in the middle cannot be split like this.

s.6.4.2. True. The service time is less variable.

s.6.4.3. Let's first check that $\rho < 1$.

```
>>> labda = 4.0
>>> ES0 = 10.0 / 60  # in hours
>>> labda_f = 1.0 / 3
>>> ER = 30.0 / 60  # in hours
>>> A = 1.0 / (1 + labda_f * ER)
>>> A
0.8571428571428571
>>> ES = ES0 / A
>>> ES
0.19444444444444445
>>> rho = labda * ES
>>> rho
0.7777777777777778
```

```
>>> Ca2 = 1.0
>>> C02 = 0.0  # deterministic service times
>>> Ce2 = C02 + 2 * A * (1 - A) * ER / ES0
>>> Ce2
0.7346938775510207
>>> EW = (Ca2 + Ce2) / 2 * rho / (1 - rho) * ES
```

```
>>> EW
0.5902777777777779
>>> EW + ES  # = EJ
0.7847222222222223
```

s.6.4.4. The time to fail is the time in between two interruptions. By assumption, the failure times are $\text{Exp}(\lambda_f)$, hence $m_f = 1/\lambda_f$. The expected duration of an interruption is $\mathsf{E}[R]$. With this

$$A = \frac{m_f}{m_f + \mathsf{E}[R]} = \frac{1/\lambda_f}{1/\lambda_f + \mathsf{E}[R]} = (1 + \lambda_f \mathsf{E}[R])^{-1}.$$

s.6.4.5. When doing the computations by hand, I worked from bottom to top. However, for the solutions, the current sequence is perhaps easier understand.

$$\begin{aligned}
\mathsf{E}[S|S_0, N] &= \mathsf{E}\left[S_0 + \sum_{i=1}^{N} R \middle| S_0, N\right] = S_0 + N\mathsf{E}[R], \text{ since } \{R_i\} \text{ are iid,} \\
\mathsf{E}[S|S_0] &= \mathsf{E}[\mathsf{E}[S|S_0, N]|S_0] = \mathsf{E}[S_0 + N\mathsf{E}[R]|S_0] = S_0 + \lambda_f S_0 \mathsf{E}[R] = S_0/A \\
\mathsf{E}[S] &= \mathsf{E}[\mathsf{E}[S|S_0]] = \mathsf{E}[S_0/A] = \mathsf{E}[S_0]/A. \\
\mathsf{V}[S|S_0, N] &= \mathsf{V}\left[S_0 + \sum_{i}^{N} R_i | S_0, N\right] = N\mathsf{V}[R], \text{ as } \mathsf{V}[S_0|S_0] = 0, \\
\mathsf{V}[S|S_0] &= \mathsf{E}[\mathsf{V}[S|S_0, N]|S_0] + \mathsf{V}[\mathsf{E}[S|S_0 N]|S_0] = \lambda_f S_0 \mathsf{V}[R] + \mathsf{V}[S_0 + N\mathsf{E}[R]|S_0] \\
&= \lambda_f S_0 \mathsf{V}[R] + (\mathsf{E}[R])^2 \mathsf{V}[N|S_0] = \lambda_f S_0 \mathsf{V}[R] + (\mathsf{E}[R])^2 \lambda_f S_0 = \lambda_f \mathsf{E}\left[R^2\right] S_0 \\
\mathsf{V}[S] &= \mathsf{E}[\mathsf{V}[S|S_0]] + \mathsf{V}[\mathsf{E}[S|S_0]] = \lambda_f \mathsf{E}\left[R^2\right] \mathsf{E}[S_0] + \mathsf{V}[S_0]/A^2.
\end{aligned}$$

s.6.4.6.

$$\begin{aligned}
C_s^2 &= \frac{\mathsf{V}[S]}{(\mathsf{E}[S])^2} = \frac{V(S)A^2}{(\mathsf{E}[S_0])^2} = \frac{\mathsf{E}\left[S_0^2\right] + \lambda_f \mathsf{E}\left[R^2\right] \mathsf{E}[S_0] A^2 - (\mathsf{E}[S_0])^2}{(\mathsf{E}[S_0])^2} \\
&= \frac{\mathsf{E}\left[S_0^2\right] - (\mathsf{E}[S_0])^2}{(\mathsf{E}[S_0])^2} + \frac{\lambda_f \mathsf{E}\left[R^2\right] \mathsf{E}[S_0] A^2}{(\mathsf{E}[S_0])^2} = C_0^2 + \frac{\lambda_f \mathsf{E}\left[R^2\right] A^2}{\mathsf{E}[S_0]}.
\end{aligned}$$

s.6.4.7. When repair times are exponentially distributed with mean $\mathsf{E}[R]$ we have that $\mathsf{E}\left[R^2\right] = 2(\mathsf{E}[R])^2$. Since $A = 1/(1 + \lambda_f \mathsf{E}[R])$,

$$\begin{aligned}
\lambda_f \mathsf{E}\left[R^2\right] A^2 &= 2\lambda_f (\mathsf{E}[R])^2 A^2 = 2\frac{\lambda_f \mathsf{E}[R]}{1 + \lambda_f \mathsf{E}[R]} A \mathsf{E}[R] \\
&= 2\left(1 - \frac{1}{1 + \lambda_f \mathsf{E}[R]}\right) A\mathsf{E}[R] = 2(1 - A)A\mathsf{E}[R].
\end{aligned}$$

s.6.4.8. Take T_k as the end of the kth repair, and $Y(t) = \int_0^t u(s)\,\mathrm{d}s$, where $u(s) = 1$ if the server is up (operational) and 0 if the server is down (broken). Thus $Y(t)$ is the total amount of time the system has been up during $[0, t]$, and $Y(t)/t$ becomes the fraction of up time (which is the availability) in the limit. As for X_k, this is $X_k = Y(T_k) - Y(T_{k-1})$, which is the up time in $[T_{k-1}, T_k]$. Therefore, in the limit, X is the average uptime between two break downs, hence $X \sim m_f$. Finally, the time between two inspections consists of an uptime and break down. Hence, the rate at which such inspection epochs occur is $1/(m_f + \mathsf{E}[R])$.

s.6.5.1. True.

s.6.5.2. False. The second station has a utilization of $80/(2*60) = 8/12 = 2/3$, while the first has a utilization of $45/60 = 3/4$, which is higher.

s.6.5.3. False. In general the number of jobs in queue is much higher than $w - N$. Consider the example $S_1 = 10$ and $N_1 = 10$, and $S_2 = 1, N_2 = 20$, and $n = 2$. Clearly, 19 machines at station 2 are always empty.

s.6.5.4. False. It is evidently wrong: the units at the LHS and RHS don't check.

s.6.5.5. First station 1.

```
>>> labda = 2.0
>>> S1 = 20.0 / 60
>>> rho1 = labda * S1
>>> ca1 = 2.0
>>> cs1 = 0.5
>>> EW1 = (ca1 + cs1) / 2 * rho1 / (1 - rho1) * S1
>>> EJ1 = EW1 + S1
>>> EJ1
1.1666666666666665
```

Now station 2. We first need to compute C_{d1}^2.

```
>>> cd1 = (1 - rho1 ** 2) * ca1 + rho1 ** 2 * cs1
>>> cd1
1.3333333333333335
>>> labda = 2
>>> S2 = 25.0 / 60
>>> rho2 = labda * S2
>>> ca2 = cd1  # here we use our formula
>>> cs2 = 0.5
>>> EW2 = (ca2 + cs2) / 2 * rho2 / (1 - rho2) * S2
>>> EJ2 = EW2 + S2
>>> EJ2
2.3263888888888897
>>> EJ1 + EJ2
3.4930555555555562
```

BIBLIOGRAPHY

S. Asmussen. *Applied Probability and Queues*. Springer-Verlag, Berlin, 2003.

M. El-Taha and S. Stidham Jr. *Sample-Path Analysis of Queueing Systems*. Kluwer Academic Publishers, 1998.

W.J. Hopp and M.L. Spearman. *Factory Physics*. Waveland Press, Inc., 3rd edition, 2008.

N. D. Van Foreest and O.A. Kilic. An intuitive approach to inventory control with optimal stopping. *European Journal of Operational Research*, 311(3):921–924, 2023.

A.A. Yushkevich and E.B. Dynkin. *Markov Processes: Theorems and Problems*. Plenum Press, 1969.

NOTATION

a_k = Number of arrivals in kth period

$A(t)$ = Number of arrivals in $[0, t]$

A_k = Arrival time of kth job

$\tilde{A}_k$ = Start of service of kth job

B = General batch size

B_k = Batch size at kth arrival time

c_k = Service/production capacity in kth period

d_k = Number of departures in kth period

c = Number of servers

C = General batch size during a service time

C_a^2 = Squared coefficient of variation of inter-arrival times

C_s^2 = Squared coefficient of variation of service times

$D(t)$ = Number of departures in $[0, t]$

D_k = Departure time of kth job

δ = Departure rate

F = Distribution of service time of a job

I = Idle time of single server

J_k = Sojourn time of kth job

$L(t)$ = Number of customers/jobs in system at time t

L_k = Number in system as end of kth period

$L_s(t)$ = Number of customers/jobs in service at time t

λ = Arrival rate

μ = Service rate

$N(t)$ = Number of arrivals in $[0, t]$

$N(s, t)$ = Number of arrivals in $(s, t]$

$p(n)$ = Long-run time average that system contains n jobs

$\pi(n)$ = Stationary probability that an arrival sees n jobs in system

$Q(t)$ = Number of customers/jobs in queue at time t

Q_k = Queue length as seen by kth job, or at end kth period

ρ = utilization of a single server

S = Generic service time

S_k = Service time of kth job

U = Busy time of single server

W = Generic waiting time (in queue)

W_k = Waiting time in queue of kth job

X = Generic inter-arrival time between two consecutive jobs

X_k = Inter-arrival time between job $k-1$ and job k