Stochastic Simulation
using Python

Version: March 9, 2023

Written by: Marko Boon,
Mark van der Boor,
Johan van Leeuwaarden,
Britt Mathijsen,
Jorn van der Pol,
Jacques Resing

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

Introduction

Mathematical modelling traditionally focussed on realistic yet tractable models. It was often the case that a simple model was favoured over a complex but more realistic one, since the latter, more complex model was harder to analyse mathematically. Nowadays, fast computers are available and more complex models can be analysed numerically. One of the approaches in this area is simulation.

In the literature, the term *simulation* may have different meanings, depending on the used technique. In all situations, however, simulation refers to the imitation of the operation of a real-world process or system over time. In these lecture notes, we discuss simulation techniques for *stochastic processes*, i.e., processes that exhibit randomness. Among these techniques are Monte Carlo simulation, discrete simulation, and discrete-event simulation. All of these simulation techniques rely on repeated random sampling to compute their results. They are widely used in different areas in mathematics and physics such as fluids, cellular structures, queueing theory and risk theory.

The theoretical background of stochastic simulation may not be very hard; however, simulation building requires some practice. These lecture notes come with many examples written in the programming language Python. A short introduction to Python, along with many examples that are particularly relevant for stochastic simulation, can be found in Appendix A.

We would like to thank Harry van Zanten and Jaron Sanders for their valuable comments and suggestions.

Marko Boon, Mark van der Boor, Johan van Leeuwaarden, Britt Mathijsen, Jorn van der Pol, Jacques Resing

Eindhoven, March 9, 2023

Contents

Part I

The basics

1

Monte Carlo simulation

Monte Carlo simulation is an approximative method, relying on repeated random sampling from probability distributions, to obtain numerical results for probabilistic problems. We will refer to The method is particularly useful when the problems are (too) difficult to solve analytically. We will use the terms Stochastic Simulation and Monte Carlo Simulation interchangeably, but some authors reserve the term Monte Carlo simulation for probabilistic algorithms for numerical integration and Monte Carlo statistical tests. In any case, the main aspect of Monte Carlo simulation is that one performs a *large* number of independent runs to estimate unknown model parameters. The name "Monte Carlo method" was suggested by physicists that developed computer simulations for the Manhattan Project, soon after World War 2, and played an important role in the development of the hydrogen bomb. The method is not only used in physics and physical chemistry, but also in (stochastic) operations research. In this chapter we discuss the basic principles of Monte Carlo simulation, starting with the theory behind the method.

1.1 Estimation of mean and tail probability

If you throw a fair coin many times, you expect that the fraction of heads will be about 50%. In other words, if we see the tossing of coins as Bernoulli experiments and we define random variables Z_i to be equal to 1 if heads occurs, and 0 if tails occurs, we expect the sample mean, $\overline{Z}_n := (Z_1 + Z_2 + \cdots + Z_n)/n$, to be close to the theoretical mean, $\mathbb{E}[Z_i] = 1/2$.

The law of large numbers. The law of large numbers states that this holds in a general setting: if $Z_1, Z_2, \ldots Z_n$ are i.i.d. random variables with mean $z := \mathbb{E}[Z_1]$ and finite variance, the probability of the *sample mean* being close to z is large. In fact, for every $\varepsilon > 0$,

$$\lim_{n\to\infty} \mathbb{P}\left(\left|\frac{Z_1 + Z_2 + \cdots + Z_n}{n} - z\right| < \varepsilon\right) = 1. \tag{1.1}$$

Now suppose that we have a method to obtain i.i.d. outcomes $Z_1, Z_2, Z_3, \ldots$, while we do not know the value of z. Think for example of a coin with an unknown probability z of throwing heads. The discussion above suggests using the sample mean

$$\overline{Z} \equiv \overline{Z}_n := \frac{Z_1 + Z_2 + \cdots + Z_n}{n} \tag{1.2}$$

as an estimator for z. This method is known as *Monte Carlo simulation* (after the famous city with many casinos). As we will see, many quantities of interest can be expressed as an expectation and

can therefore be estimated using Monte Carlo simulation.

First we will define some terminology that will often be used in these lecture notes. Each of the random variables above is called a *replication*, or the result of a *run*. The number n that appears in Equation (1.2) is called the *number of independent runs*, or the *number of independent replications*.

Example 1.1 (Estimation of mean and tail probability)**.** Consider $X \sim \text{Exp}(10)$ and suppose that we are interested in the expected value of X. We know, of course, that $\mathbb{E}[X] = 0.1$, but this is a nice example to illustrate the Monte Carlo technique. If we are able to simulate $X_1, X_2, \ldots, X_n \sim \text{Exp}(10)$, it is intuitively clear that we can use the sample mean as an estimator for $\mathbb{E}[X]$ and that this estimate will be better if n gets larger.

In Figure 1.1, fifteen estimates are plotted for each number of runs in $20, 40, 60, \ldots, 2000$. As you can see, the spread among the estimates decreases as the number of runs increases. Hence, estimates become better as the number of runs they are based on increases. See Listing 1.1 for the Python code we used to estimate $\mathbb{E}[X]$ based on a fixed number of runs.

Listing 1.1: Estimating $\mathbb{E}[X]$.

```
1  from numpy import mean
2  from scipy import stats
3
4  expDist = stats.expon(scale=1/10)    # Exp. dist. with parameter 10
5  n = 100000                           # Number of independent replications
6  z = expDist.rvs(n)                   # Generate n EXP(10) random variates
7  est = mean(z)                        # Calculate estimate of expectation
8  print(est)
```

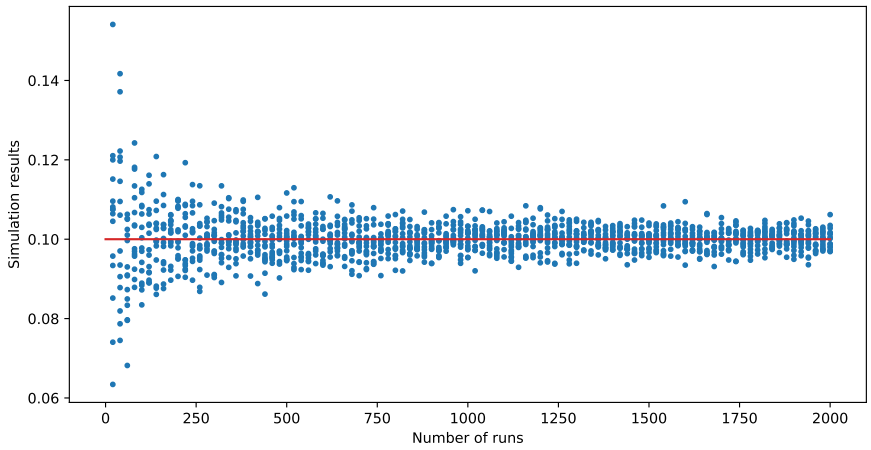


Figure 1.1: Plot of the spread in estimators against the number of independent runs they are based on. The true expected value (0.1) is plotted in red.

It is also possible to estimate the probability that $X > 0.15$. If we denote by $\mathbf{1}_{\{\cdot\}}$ the indicator function, that is, $\mathbf{1}_{\{A\}} = 1$ if A holds, and $\mathbf{1}_{\{A\}} = 0$ otherwise, then $\mathbf{1}_{\{A\}}$ is a Bernoulli random variable and $\mathbb{E}\left[\mathbf{1}_{\{X>0.15\}}\right] = \mathbb{P}(X > 0.15)$. Hence, if we let $Y_i = \mathbf{1}_{\{X_i>0.15\}}$, the sample mean $\overline{Y} = (Y_1 + Y_2 + \cdots + Y_n)/n$ is an estimator for $\mathbb{P}(X > 0.15)$. The source code for this situation can be found in Listing 1.2.

Remark: To automatically convert the booleans True and False to, respectively, 1 and 0, we need to import the `mean` function from the `numpy` package. The equivalent function from the `statistics`

Listing 1.2: Estimating $\mathbb{P}(X > 0.15)$.

```
y = (z > 0.15)                  # Replace all entries > 0.15 by TRUE
est = mean(y)                   # and entries <= 0.15 by FALSE.
print(est)                      # Then convert TRUE -> 1, FALSE -> 0
exact = 1 - expDist.cdf(0.15)   # Compare to the theoretical value
print(exact)
```

package does not support this functionality.

Exercise 1.1. Suppose that $X_1, X_2, \ldots, X_{10}$ are independently distributed according to the normal distribution with mean $\mu = 1$ and variance $\sigma^2 = 4$. Use Monte Carlo simulation to estimate the mean and the variance of $X_1 + X_2 + \cdots + X_{10}$.

Solution:

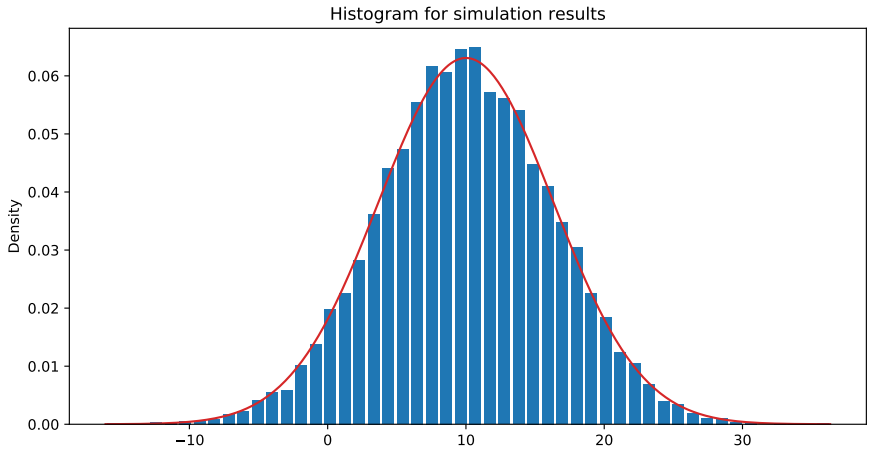


Figure 1.2: Histogram based on 10,000 independent simulation runs. The theoretical density is plotted in red.

Sample source code can be found in Listing 1.3.

Listing 1.3: Estimating $\mathbb{E}[X_1 + X_2 + \cdots + X_{10}]$ in Exercise 1.1.

```
from numpy import zeros, mean, var
from scipy import stats

n = 10000                       # Number of runs
mu = 1                          # The expectation
sigma = 2                       # The standard deviation
result = zeros(n)               # Vector to store the results
dist = stats.norm(mu, sigma)    # The normal distribution

for i in range(n):
    rnd = dist.rvs(10)          # Sample 10 random numbers
    result[i] = sum(rnd)        # Take the sum and store the result

print(mean(result))
print(var(result))
```

Since $X_1 + X_2 + \cdots + X_{10}$ is the sum of ten random variables with mean 1, its expected value is 10. Figure 1.2 depicts a histogram based on 10,000 independent simulation runs.

As you can see, the values are centered around the value 10, but some runs yield a result that is much smaller or larger than 10. The variance of ten independent random variables, is the sum of the individual variances, which is equal to 40 in this case.

Exercise 1.2. Let Y be uniformly distributed on the random interval $[0, X]$, where X follows an exponential distribution with parameter $\lambda = 1$. Use simulation to calculate $\mathbb{E}[Y]$ and $\mathbb{P}(Y > 1)$.

Solution: In each run of the simulation, two random variables have to be drawn. We first draw X, which is distributed according to the exponential distribution with rate λ. If x is the realization of X, in the next step the random variable Y has to be drawn from the uniform distribution on $[0, x]$. Repeating this algorithm a suitable number of times and taking the average of all the outcomes, yields an estimate for $\mathbb{E}[Y]$.

The quantity $\mathbb{P}(Y > 1)$ can be calculated in much the same way, but here we use as estimator Z, where $Z = 1$ if $Y > 1$ and $Z = 0$ otherwise. See Listing 1.4.

Listing 1.4: Estimating $\mathbb{E}[Y]$ and $\mathbb{P}(Y > 1)$ using Monte Carlo simulation.

```
lam = 1                              # The parameter lambda
result = zeros(n)                    # Vector to store the results
expDist = stats.expon(scale=1/lam)   # The Exp(lambda) distribution

for i in range(n):
    x = expDist.rvs(1)               # Sample random lambda
    uDist = stats.uniform(0, x)      # The uniform distribution
    result[i] = uDist.rvs(1)         # Sample 1 random uniform number

print(mean(result))
z = (result > 1)                     # indicator variable 1_{Y > 1}
print(mean(z))
```

The true values of $\mathbb{E}[Y]$ and $\mathbb{P}(Y > 1)$ can be calculated by conditioning on X. We find

$$\mathbb{E}[Y] = \mathbb{E}[\mathbb{E}[Y \mid X]] = \mathbb{E}\left[\frac{1}{2}X\right] = \frac{1}{2}; \tag{1.3}$$

and

$$\mathbb{P}(Y > 1) = \int_{x=0}^{\infty} \mathbb{P}(Y > 1 \mid X = x)\, f_X(x)\mathrm{d}x = \int_{x=1}^{\infty} \frac{x-1}{x}\mathrm{e}^{-x}\mathrm{d}x, \tag{1.4}$$

since $\mathbb{P}(Y > 1 \mid X = x) = 0$ if $x < 1$ and $\mathbb{P}(Y > 1 \mid X = x) = (x-1)/x$ otherwise. The integral on the right cannot be calculated explicitly (which may be a reason to use simulation!), but it is approximately equal to 0.1484.

	True value	**Result 1**	**Result 2**	**Result 3**	**Result 4**	**Result 5**
$\mathbb{E}[Y]$	0.5	0.4977	0.5060	0.4982	0.4944	0.5011
$\mathbb{P}(Y > 1)$	≈ 0.1484	0.1458	0.1516	0.1488	0.1493	0.1501

Table 1.1: True values and simulation results for $\mathbb{E}[Y]$ and $\mathbb{P}(Y > 1)$, based on 10,000 independent runs. We see that the true values are closely approximated.

Various estimates, based on 10,000 independent runs, can be found in Table 1.1.

Remark: Creating a random variable object from the Python scipy.stats library, and sampling *one* number at a time, is extremely slow. It is much more efficient to sample multiple values in one call of the `rvs` function. See Appendix B for a discussion on this topic and one way to solve this issue. There is an alternative, much more efficient solution where the uniform distribution is created with a vector of n random samples of X. However, this solution is arguably more dangerous, because not every programming language supports this construction, and its behaviour may be unexpected.

1.2 Various simple probabilistic problems

Example 1.2. The birthday problem is the seemingly paradoxical result that in a group of only 23 people, the probability that two people share the same birthday is approximately 50%. In this example we use simulation to show that this is indeed the case. We will write a simulation to estimate the probability that in a group of n people, at least two people share the same birthday.

The birthday of the i-th person is a discrete uniform random variable on $0, 1, 2, 3, \ldots, 364$ (we ignore leap years). In each run of the simulation, n such random variables have to be drawn. Let Z be a random variable that takes on the value 1 if at least two of the birthdays are the same and 0 otherwise. Then the probability of at least two people sharing their birthday is $\mathbb{E}[Z]$, and we find that Z is our estimator.

Sample source code for this exercise can be found in Listing 1.5. From Figure 1.3, it can be seen that the probability that in a group of 23 people, at least two of them share the same birthday, is about 50%.

Listing 1.5: Sample code for the birthday problem simulation.

```
1  rng = random.default_rng()    # The random number generator
2
3  def sameBirthday(birthdays) :
4      cal = [False] * 365       # create a list of 365 booleans 'False'
5      for b in birthdays :
6          if not cal[b]:        # no birthday on this date (so far)
7              cal[b] = True
8          else :                # someone else has the same birthday
9              return True
10     return False              # so name birthday was found
11
12 n = 20                        # The number of people
13
14 nrRuns = 100000               # Multiple runs
15 counter = 0
16 for i in range(nrRuns) :
17     birthdays = rng.integers(0,365, size=n)
18     if sameBirthday(birthdays) :
19         counter += 1
```

Example 1.3. Two players play the following game. A coin is tossed $N = 10$ times. If a head occurs, player A receives one point, and otherwise player B receives one point. What is the probability that player A is leading at least 60% of the time?

The main ingredient of our simulation will be the simulation of a coin toss. Recall that tossing a coin is essentially drawing a Bernoulli random variable with parameter $p = 0.5$.

Let A_n, resp. B_n, be the number of times player A, resp. player B, received a point before or at the n-th toss and define $D_n = A_n - B_n$. Note that $D_n > 0$ if and only if player A is leading at time n.

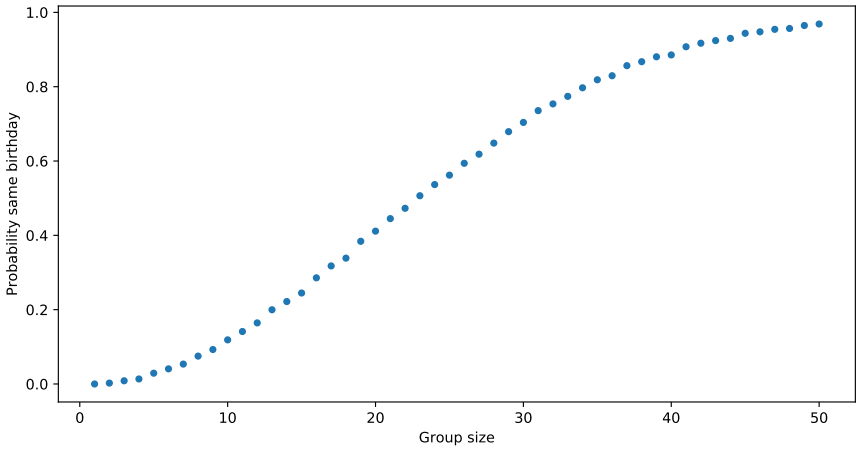


Figure 1.3: Sample output for the birthday problem. Here, the size of the group, n, varies between 2 and 50. The number of independent replications is 10,000.

Now if D_n is known, and we toss the coin for the $n+1$-st time, D_n either increases by one (player A receives a point) or decreases by one (player B receives a point). This can be used to write a simulation.

As an estimator, we will use Z, where $Z = 1$ if $D_n > 0$ for at least six different n and $Z = 0$ otherwise. See Listing 1.6.

We find that the probability that A is leading at least 60% of the time is approximately 0.38.

Listing 1.6: Source code for simulation of the coin tossing game in Example 1.3.

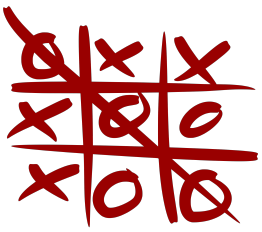
```
from scipy import stats
from numpy import zeros, cumsum, mean

nrTosses = 100          # number of tosses per run
runs = 10000            # number of runs
results = zeros(runs)
for i in range(runs) :
    tosses = stats.bernoulli(0.5).rvs(nrTosses)
    Awins = tosses      # if toss = 1, A wins; if toss = 0, B wins
    Bwins = 1 - tosses
    An = cumsum(Awins)  # cumulative number of points player A
    Bn = cumsum(Bwins)  # cumulative number of points player B
    Dn = An - Bn        # difference
    # check if player A is leading at least 60% of this run:
    results[i] = (sum(Dn > 0) >= 0.60*nrTosses)

print(mean(results))    # Probability that player A is leading at least 60% of the time
```

1.3 Strategic turn based games

The previous examples have shown that Monte Carlo simulation can be a useful tool to evaluate games of chance. This becomes particularly interesting when the games involve strategic behavior. Even for games that do not involve randomness, Monte Carlo simulation can provide some useful insights. In this section, as an example, we discuss the game of Tic-tac-toe.



Tic-tac-toe is a famous simple game for two players, played on a 3×3 board. Players take turns to place their mark on the board. The goal is to place three marks in a (horizontal, vertical, or diagonal) row. In principle, this is an uninteresting game, because it is well-known that there exists a strategy that ensures that the player will never lose regardless of whether they begin or not. However, it is a popular game among children unaware of this optimal strategy. We will use this game as an example to illustrate how stochastic simulation can be used to evaluate strategies to play such a game. In particular we will introduce the following two strategies for a player:

- RANDOM: the player places their mark uniformly at random in one of the empty squares (i.e. each square with equal probability);
- SMART: the player will first check each of the empty squares to find out if placing their mark in that square will result in a win. If not, the player will select one of the empty squares uniformly at random.

Note that these are two extremely simple strategies, as the former (RANDOM) does not look ahead at all, while the latter looks ahead only one step. Still, it is interesting to implement the Tic-tac-toe game with these two strategies and find out how much better the SMART strategie is than the RANDOM strategie. Additionally, we can use simulation to answer the question if the probability of winning the game is greater for Player 1 than for Player 2. We assume that assume that Player 1 (with mark 'X') always makes the first move in a game.

Before we start implementing the strategies, we need to program a data structure for the board. Because of the grid structure, it makes sense to use a 3×3 matrix with values 0 (the square is empty), 1 (Player 1 placed an 'X' in the square), or 2 (Player 2 placed an 'O' in the square). We explain the code, which is shown in Listing 1.7, in more detail below.

- It is always a good habit to define constants instead of working with specific numeric values in the code. For example, we have defined constants for the state of a game (still in progress, win player 1, win player 2, draw), for the size of the board (which is three by three, but the code will work for larger square boards too).
- The function `copy()` is essential in games like these! When evaluating strategies, as discussed later, we will need to simulate a particular move and predict the outcome of the game depending on that move. This requires making a copy of the board every time, before the move is made.
- The function `getPossibleMoves()` returns a matrix with two columns. Each row in this matrix corresponds to an empty square, represented by the row number (0, 1, 2) and the column number (0, 1, 2) of the square.
- The function `getPlayerTurn()` returns the number of the player (1 or 2) whose turn it is. This is simply determined by counting the number of moves so far, which is the difference between n^2 (for an $n \times n$ board) and the number of possible moves. Since the turns alternate, we have to take this number modulo 2. Finally, we add 1, because we have players 1 and 2, rather than 0 and 1 (and because player 1 starts the game).
- The function `makeMove(row, column)` makes a move on behalf of the player whose turn it is, in the square represented by the specified row and column number.
- The function `evaluate()` evaluates the board and determines the current state, which is one of the aforementioned four possible states of a game (still in progress, win player 1, win player 2, draw). There are multiple ways to implement this, but given that we want this function to work for an arbitrarily large square board, we compute the *products* of the rows, columns, and both

diagonals of all board entries. If one of these products is equal to p^n, where $p \in \{1, 2\}$ and n is the size of the board (so 3 by default), we know that player p has won. Note that $1^n = 1$ so a product equal to 1 corresponds to a win of player 1. If none of the players has won, the number of empty squares is counted. If this is equal to zero, the game has ended in a draw; otherwise the game is still in progress.

The next challenge is to implement the two strategies. The Python code can be found in Listing 1.8. The most important function is called `nextMove(board, strategy)`, which returns the next move that is made by a player who plays according to the specified strategy, on the given board. The first strategy, RANDOM, is extremely simple to implement: The player will simply select one of the possible moves uniformly at random. The implementation of the SMART strategy, shown in lines $13 - 24$ of Listing 1.8, is slightly more complicated, which is exactly the reason for us to include this example in these lecture notes. First, we determine whose turn it is. Next, we iterate of all possible moves. For each possible move, we make this move in *a copy of* the given board. After making this move, we evaluate this copied board. If the move results in a win for the player who made the move, this move is returned (because no other move can be better than this move). If the move does not result in a win, the search for the optimal move is continued. If no move results in a win for the player, a random move is made.

Using this function `nextMove(board, strategy)`, a game between two players can be played. This is done in the function `playGame(board, strategyPlayer1, strategyPlayer2)`, which is relatively straightforward now. Given a specified initial board setup, in a while loop, it is determined whose turn it is and according to this player's strategy, the next move is made. This is continued until the game finishes. Then the function returns the outcome of the game (i.e. who won) and the final board.

Obviously, to get reliable results, we need to play a large number of games. In the next chapter, we will discuss exactly how to determine how many simulation runs are required. For now, we selected 100,000 runs and compared all four combinations of the two players having each of the two strategies. The results, shown in the table below, confirm that player 1 clearly has an advantage of being allowed to make the first move, and that the SMART strategy outperforms the RANDOM strategie. As such, it is particularly interesting to see what happens if player 1 plays RANDOM while player 2 plays SMART. In this combination, player 2 has a higher probability of winning that player 1, compensating the disadvantage of not being allowed to make the first move.

	Player 1 RANDOM	Player 1 SMART
Player 2 RANDOM	P(Player 1 wins) = 0.59 P(Player 2 wins) = 0.29 P(Draw) = 0.13	P(Player 1 wins) = 0.81 P(Player 2 wins) = 0.12 P(Draw) = 0.07
Player 2 SMART	P(Player 1 wins) = 0.40 P(Player 2 wins) = 0.52 P(Draw) = 0.08	P(Player 1 wins) = 0.69 P(Player 2 wins) = 0.27 P(Draw) = 0.04

It is also interesting to see how the probabilities of winning the game depend on the first move. This can easily be done by creating a new board, placing the first move at a certain position, and then letting the continue using the `playGame(board, strategyPlayer1, strategyPlayer2)` function. Although we omit the specific numerical results here, it turns out that when both players play according to the SMART strategy, the probability of winning after having made the first move in the center is higher (0.85) than when making the first move in a corner (0.73) or on one of the four side squares (0.61).

To conclude this section, we note that much literature is available on Monte Carlo simulation techniques for strategic games. In fact, when combining Monte Carlo simulation with more sophisticated search algorithms, it turns out that it is actually capable of finding (with high probability) the optimal move for simple games such as Tic-tac-toe. For more complicated games like Go, there is no way of

determining the optimal strategy (yet?), but Monte Carlo plays a crucial role in evaluating the various strategies. We encourage the interested reader to read more about the various Artificial Intelligence (AI) algorithms involving random sampling, in particular the Monte Carlo Tree Search (MCTS) algorithm, and how it can be applied to a game of Tic-tac-toe.

1.4 Other applications of Monte Carlo simulation

Monte Carlo simulation can be used for more than just the analysis of stochastic systems. One particularly nice example is the estimation of π.

Example 1.4 (Estimation of π)**.** Consider the square with $(\pm 1, \pm 1)$ as its corner points and the inscribed circle with centre $(0, 0)$ and radius 1 (see Figure 1.4). The area of the square is 4 and the area of the circle is π. If we choose a point at random in the square, it falls within the circle with probability $\pi/4$. Hence, if we let $Z_i = (X_i, Y_i)$, with $X_i, Y_i \sim \text{Uniform}\,(-1, 1)$ independently and if we define the sequence $C_i = \mathbf{1}_{\{X_i^2 + Y_i^2 \leq 1\}}$, then the sample mean of the C_i converges (in probability) to $\pi/4$:

$$\widehat{p} := \frac{C_1 + C_2 + \cdots + C_n}{n} \xrightarrow{p} \frac{\pi}{4}. \tag{1.5}$$

We find that $\widehat{\pi} := 4\widehat{p}$ is an estimator for π.

Python source code for this example can be found in Listing 1.9.

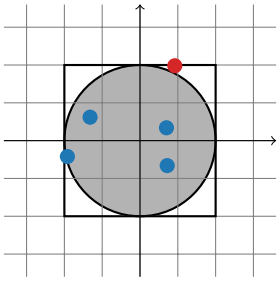


Figure 1.4: The setting used for the hit-and-miss estimator for π, with five random points.

For obvious reasons the sample average of the C_i is often called the hit-or-miss estimator.

Example 1.5 (Numerical integration)**.** Another nice application of Monte Carlo simulation is numerical integration. Recall that integrating some function g over an interval $[a, b]$ yields a number that can be interpreted as $(b - a)$ multiplied by the "mean value" of g on $[a, b]$. This can be written in terms of the expected value operator:

$$\int_{x=a}^{b} g(x) \frac{1}{b-a} \mathrm{d}x = \mathbb{E}\left[g(U)\right], \tag{1.6}$$

where U is uniformly distributed on (a, b).

Exercise 1.3. If $U_1, U_2, \ldots, U_n$ are independent and uniformly distributed on $(0, 1)$, show that

$$Z := \frac{g(U_1) + g(U_2) + \cdots + g(U_n)}{n}$$

is an unbiased estimator for

$$z := \int_{x=0}^{1} g(x) \mathrm{d}x.$$

Solution: Let $U \sim \text{Uniform}(0,1)$. The density of U is then $f(x) = 1$ for $0 < x < 1$ and $f(x) = 0$ otherwise. We find that

$$\mathbb{E}[g(U)] = \int_{x=0}^{1} g(x)f(x)\mathrm{d}x = z. \tag{1.7}$$

From this, it follows immediately that Z is an unbiased estimator for z.

Exercise 1.4. We know that

$$\int_{x=0}^{1} 4\sqrt{1-x^2}\mathrm{d}x = \pi.$$

We will now use numerical integration to estimate the value of π and compare the results to the results obtained via the hit-or-miss estimator.

It follows from the previous exercise that an unbiased estimator for $\int_{x=0}^{1} 4\sqrt{1-x^2}\mathrm{d}x$ is given by

$$\frac{1}{n}\sum_{i=1}^{n} 4\sqrt{1-U_i^2}, \tag{1.8}$$

with $U_1, U_2, \ldots, U_n$ independent and uniformly distributed on $(0,1)$. Sample code, based on this observation, can be found in Listing 1.10.

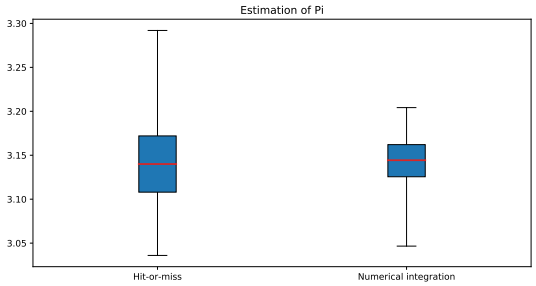


Figure 1.5: Comparison of results of two simulation algorithms, hit-or-miss vs. numerical integration.

For a comparison of the two estimators, see Figure 1.5, which depicts the results of 200 repetitions of both algorithms, each repetition using 1,000 independent realizations. From this plot, it is clear that the numerical integration algorithm performs better than the hit-or-miss estimator, since the estimates produced by numerical integration algorithm are much more concentrated around the true value π. However, this does by no means imply that numerical integration is always favorable over the hit-or-miss estimator. For example, try to integrate the following integral numerically:

$$\int_{x=0}^{1} \frac{2}{\sqrt{1-x^2}}\mathrm{d}x = \pi. \tag{1.9}$$

The quality of the numerical estimator based on this integral is much less than the quality of the hit-or-miss estimator.

Exercise 1.5. Use Monte Carlo integration to evaluate the integral

$$\int_{x=0}^{2} \mathrm{e}^{-x}\mathrm{d}x. \tag{1.10}$$

Solution: There are various ways in which a Monte Carlo algorithm for calculating this integral can be implemented:

(i) The most straightforward way, perhaps, is to write the integrand as $2e^{-x} \times \frac{1}{2}$, and to recognise the density of the Uniform $(0, 2)$-distribution. Generating Uniform $(0, 2)$-random numbers $U_1, U_2, \ldots, U_n$, calculating $Z_i = 2e^{-U_i}$ and then setting $(Z_1 + Z_2 + \cdots + Z_n)/n$ as an estimator for the integral.

(ii) You could also identify the integral as the probability that $X \sim \text{Exp}(1)$ is at most 2. In fact,

$$\int_{x=0}^{2} e^{-x} dx = \int_{x=0}^{\infty} \mathbf{1}_{\{x \leq 2\}} e^{-x} dx = \mathbb{P}(X \leq 2). \tag{1.11}$$

Hence, we could draw Exp (1)-random numbers $X_1, X_2, \ldots, X_n$, put $Z_i = \mathbf{1}_{\{X_i \leq 2\}}$ and use $(Z_1 + Z_2 + \cdots + Z_n)/n$ as an estimator for the integral.

Both methods will produce an estimate that lies around 0.865. Using the methods that we will discuss in Chapter 2, you will be able to show that the second method produces an estimator of higher quality.

The following exercise shows that Monte Carlo integration also works for higher dimensional integrals.

Exercise 1.6. Use Monte Carlo integration to calculate the double integral

$$\iint_A x^2 y \, dy \, dx, \tag{1.12}$$

where A is the triangle with corner points $(0, 0)$, $(1, 0)$ and $(1, 3)$.

Solution: In order to integrate over the triangular area, we need to generate random points in this area. We will describe a nice method to do this. Note first that (x, y) is a point in A if and only if $0 \leq x \leq 1$ and $0 \leq y \leq 3x$. We will generate n points (X, Y) in the rectangle R, which has corner points $(0, 0)$, $(1, 0)$, $(1, 3)$, and $(0, 3)$. This can be done by choosing X uniformly between 0 and 1 and choosing Y uniformly between 0 and 3. Now, note the following:

- Each point is either in A, or in $B := R \setminus A$. Moreover, due to symmetry, B has the same shape (and same area) as A.
- $1 - X$ and $3 - Y$ have the same uniform distributions as, respectively, X and Y.

We exploit this symmetry, by replacing each point (X, Y) in B by $(1 - X, 3 - Y)$, which is in A.

An implementation of this algorithm can be found in Listing 1.11. If we use this method to calculate the integral, we must first note that the density of the uniform distribution over A is identically $\frac{2}{3}$. Our estimator therefore will be $\frac{3}{2} X^2 Y$, with (X, Y) distributed as above. A correct implementation will yield an estimate close to the true value of 0.9.

Listing 1.7: The Tic-tac-toe board for the example in Section 1.3.

```
from numpy import zeros, sum, diag, prod, append, where, transpose, random

class Board :

    # Define some constants
    SIZE = 3            # Size of the board: 3 x 3
    NOTFINISHED = 0;    # Game is not finished (yet)
    PLAYER1WINS = 1;    # Game ends in a win for player 1
    PLAYER2WINS = 2;    # Game ends in a win for player 2
    DRAW = 3;           # Game ends in a draw

    def __init__(self):  # Construct an empty 3 x 3 board
        self.board = zeros((self.SIZE, self.SIZE))

    def copy(self):      # Create a copy of the current board
        c = Board()
        c.board = self.board.copy()
        return c

    def getPossibleMoves(self):  # get a list of all possible moves (empty squares)
        moves = (self.board == 0)
        return transpose(where(moves)) # the coordinates of the empty places

    def getPlayerTurn(self):     # Return the player whose turn it is (1 or 2)
        nrSquares = self.SIZE**2
        nrEmptySquares = len(self.getPossibleMoves())
        nrMoves = nrSquares - nrEmptySquares
        return (nrMoves % 2) + 1

    def makeMove(self, row, col): # make a move at the specified square
        self.board[row, col] = self.getPlayerTurn()

    def evaluate(self):
        rowProds = prod(self.board, axis=1)
        colProds = prod(self.board, axis=0)
        diag1prod = prod(diag(self.board))
        diag2prod = prod(diag(self.board[::-1]))
        prods = append(rowProds, colProds)
        prods = append(prods, [diag1prod, diag2prod])
        if (sum(prods == 1) > 0):
            return self.PLAYER1WINS
        elif (sum(prods == 2**self.SIZE) > 0):
            return self.PLAYER2WINS
        else:
            if len(self.getPossibleMoves()) == 0:
                return self.DRAW
            else:
                return self.NOTFINISHED

    def __str__(self):
        s = ""
        chars = ["_", "X", "O"] # player 1 = X, player 2 = O
        for i in range(self.SIZE):
            for j in range(self.SIZE):
                s += chars[int(self.board[i,j])] + " "
            s += "\n"
        return s
```

Listing 1.8: The implementation of the strategies for a game of Tic-tac-toe in Section 1.3.

```
from ch1.tictactoe.Board import Board
from numpy import zeros, transpose, where, random, mean

rng = random.default_rng()    # The random number generator


STRATEGY_RANDOM = 0
STRATEGY_SMART = 1

# return the next move, according to the specified strategy
def nextMove(board, strategy):
    moves = board.getPossibleMoves()
    if strategy == STRATEGY_RANDOM:  # Random move
        return moves[rng.choice(len(moves))]
    else:       # Smart
        player = board.getPlayerTurn()
        for i in range(len(moves)):
            m = moves[i]
            b = board.copy()
            b.makeMove(m[0], m[1])
            s = b.evaluate()
            if ((s == Board.PLAYER1WINS and player == 1)
                or (s == Board.PLAYER2WINS and player == 2)):
                return m  # do the move, because the player will win
        # apparently, there was no move that results in a direct win
        # so do a random move
        return moves[rng.choice(len(moves))]

# Play one game, starting on the specified board,
# with the two player strategies.
def playGame(board, strategyPlayer1, strategyPlayer2):
    score = board.evaluate()
    while (score == Board.NOTFINISHED):  # play as long as the game is not finished
        player = board.getPlayerTurn()
        if player == 1:
            move = nextMove(board, strategyPlayer1)
        else:  # player 2
            move = nextMove(board, strategyPlayer2)
        board.makeMove(move[0], move[1])
        score = board.evaluate()
    return (score, board)


# Test the board class
b = Board()
print("Start of the game, player's turn: %i" % b.getPlayerTurn())
b.makeMove(1, 1)
b.makeMove(0, 0)
b.makeMove(1, 0)
b.makeMove(2, 2)
print(b)
finishStr = ["NOT FINISHED", "PLAYER 1 WINS", "PLAYER 2 WINS", "DRAW"]
print(finishStr[b.evaluate()])
print("Possible moves: " + str(b.getPossibleMoves()))
b.makeMove(1, 2)
print(b)
print(finishStr[b.evaluate()])

# Play one game of two players playing randomly
(score, board) = playGame(Board(), STRATEGY_RANDOM, STRATEGY_RANDOM)
print(finishStr[score])
print(board)

# Do multiple runs
nrRuns = 100000
simResults = zeros(nrRuns)
for i in range(nrRuns):
    sim = playGame(Board(), STRATEGY_RANDOM, STRATEGY_RANDOM)
    simResults[i] = sim[0]

print('Strategy: Random vs. Random')
```

Listing 1.9: Estimating the value of π using Monte Carlo integration.

```
from numpy import mean, std, sqrt
from scipy import stats

def piSim(runs):
    uDist = stats.uniform(-1, 2) # uniform on [-1, 1] !!!
    x = uDist.rvs(runs)
    y = uDist.rvs(runs)
    results = x**2 + y**2 <= 1   # Check which points are inside the unit circle
    results = 4 * results        # pi = E[4X] where X is Bernoulli distributed
    return results

n = 1000000
sim = piSim(n)

m = mean(sim)                    # estimate for pi
print(m)
```

Listing 1.10: Numerical integration, sample source code for exercise 1.4.

```
runs = 1000000              # Number of independent runs
uDist = stats.uniform(0, 1) # uniform on [0, 1]
x = uDist.rvs(runs)
results = 4 * sqrt(1 - x**2)
print(mean(results))
```

Listing 1.11: Source code for the numerical integration exercise 1.6.

```
runs = 1000000
xDist = stats.uniform(0, 1) # uniform on [0, 1]
yDist = stats.uniform(0, 3) # uniform on [0, 3]
x = xDist.rvs(runs)
y = yDist.rvs(runs)
notInA = y > 3*x            # check if point is in B
x[notInA] = 1 - x[notInA]   # move point to A
y[notInA] = 3 - y[notInA]
est = 1.5 * x**2 * y
print(mean(est))
```

2

Output analysis

In this chapter we use the theory of the central limit theorem to construct confidence intervals for the parameters that we want to estimate using simulation. These can be used to assess the simulation accuracy, or to determine the minimum required number of runs.

2.1 Confidence intervals

From now on, the setting will be as follows: we are interested in estimating a certain quantity z, and we have a random variable Z, such that $z = \mathbb{E}[Z]$.

Given independent realisations $Z_1, Z_2, \dots, Z_n$ of the random variable Z, we already know that the sample mean $\overline{Z} := (Z_1 + Z_2 + \dots + Z_n)/n$ is an unbiased estimator of z. Although this is interesting in itself, we are often also interested in the quality of the estimator: if there is a large spread in the outcomes Z_i, it is quite possible that the estimator $\overline{Z}$ is far off from the true value z.

We will often use *confidence intervals* centered around the estimator $\overline{Z}$ to give an estimate that expresses in some sense the spread of the estimate. A confidence interval is an interval in which we can assert that z lies with a certain degree of confidence.

Mathematically speaking, a $100(1-2\alpha)\%$ confidence interval is a random interval (usually depending on the outcomes Z_i), such that the probability that the interval covers the true value of z equals $1-2\alpha$. In these lecture notes, we will focus on approximate confidence intervals, inspired by the central limit theorem.[1]

2.2 The Central Limit Theorem

Consider a sequence of i.i.d. random variables $Z_1, Z_2, \dots, Z_n$, with common mean z and variance σ^2. We have already introduced the sample mean $\overline{Z}$. We will now define the sample variance S^2:

Definition 2.1 (Sample variance)**.** The quantity S^2, defined by

$$S^2 := \frac{1}{n-1} \sum_{i=1}^{n} (Z_i - \overline{Z})^2, \tag{2.1}$$

is called the *sample variance* of the random sample $Z_1, Z_2, \dots, Z_n$.

[1] Better confidence intervals can be constructed using Student's t-distribution, see e.g. Wikipedia [WikiA].

The following exercise justifies the name of the sample variance:

Exercise 2.1. Show that the sample variance is an unbiased estimator of σ^2. That is, show that

$$\mathbb{E}\left[S^2\right] = \sigma^2. \tag{2.2}$$

Solution: Note that

$$\sum_{i=1}^{n}(Z_i - \overline{Z})^2 = \sum_{i=1}^{n}(Z_i^2 - 2Z_i\overline{Z} + \overline{Z}^2) = \sum_{i=1}^{n} Z_i^2 - 2n\overline{Z}^2 + n\overline{Z}^2 = \sum_{i=1}^{n} Z_i^2 - n\overline{Z}^2. \tag{2.3}$$

From this, it follows that

$$\begin{aligned}
\mathbb{E}\left[S^2\right] &= \frac{1}{n-1}\left(\sum_{i=1}^{n}\mathbb{E}\left[Z_i^2\right] - n\mathbb{E}\left[\overline{Z}^2\right]\right) = \\
&= \frac{n}{n-1}\left(\mathbb{E}\left[Z_1^2\right] - \mathbb{E}\left[\overline{Z}^2\right]\right) \\
&= \frac{n}{n-1}\left(\mathrm{Var}\left[Z_1\right] + \mathbb{E}\left[Z_1\right]^2 - \mathrm{Var}\left[\overline{Z}\right] - \mathbb{E}\left[\overline{Z}\right]^2\right) \\
&= \frac{n}{n-1}\left(\sigma^2 + z^2 - \frac{\sigma^2}{n} - z^2\right) \\
&= \sigma^2,
\end{aligned} \tag{2.4}$$

which is what had to be shown.

The *Central Limit Theorem*, often abbreviated as CLT, relates the sample mean of a sufficiently large number of random variables to the normal distribution.

Theorem 2.1 (Central Limit Theorem)**.** *Let the sequence of random variables Y_n be defined by*

$$Y_n := \sqrt{n}\frac{\overline{Z} - z}{\sqrt{\sigma^2}}. \tag{2.5}$$

Then Y_n converges in distribution to a standard normal random variable, or

$$\lim_{n\to\infty}\mathbb{P}\left(Y_n \leq y\right) = \Phi(y), \tag{2.6}$$

where $\Phi(\cdot)$ is the standard normal cdf, given by

$$\Phi(y) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{y} e^{-\frac{1}{2}t^2}dt. \tag{2.7}$$

This theorem is often interpreted as

$$\mathbb{P}\left(Y_n \leq y\right) \approx \Phi(y) \tag{2.8}$$

and we will use it to construct approximate confidence intervals.

Note that Y_n depends on an unknown parameter, σ^2. Fortunately, by a theorem known as Slutsky's theorem, the central limit theorem still holds if we replace σ^2 by the sample variance, S^2, so that the quantity

$$Y_n := \sqrt{n}\frac{\overline{Z} - z}{\sqrt{S^2}}, \tag{2.9}$$

still converges in distribution to a standard normal random variable.

Example 2.1. This example shows intuitively, using Monte Carlo simulation, how the CLT works. The source code can be found in Listing 2.1. We generate 1 realisation from each of the i.i.d. random variables $Z_1, \ldots, Z_n$. Feel free to replace the uniform distribution by *any* other probability distribution. Using these n realisations, we can compute $\overline{Z}$. In order to compute Y_n using Equation (2.5), we need to specify the mean z and variance σ^2 of Z_i. Obviously, for the given uniform distribution we have $z = 1/2$ and $\sigma^2 = 1/12$. However, in this experiment we have chosen to use accurate Monte Carlo simulations (10^6 runs) to estimate z and σ, rather than specifying the exact values. In Listing 2.1 we simulate *nrRuns* realisations for Y_n given the specified value of n. We can plot a histogram of these simulated values and compare it to the density of the standard normal distribution. Figure 2.1 shows how fast this process converges. Already for $n = 5$ the resemblance is striking! As an exercise, you can replace σ^2 by the sample variance S^2 and verify that it still converges to a standard normal distribution. In the Python code you should replace `sigma` by `std(Zsim)`.

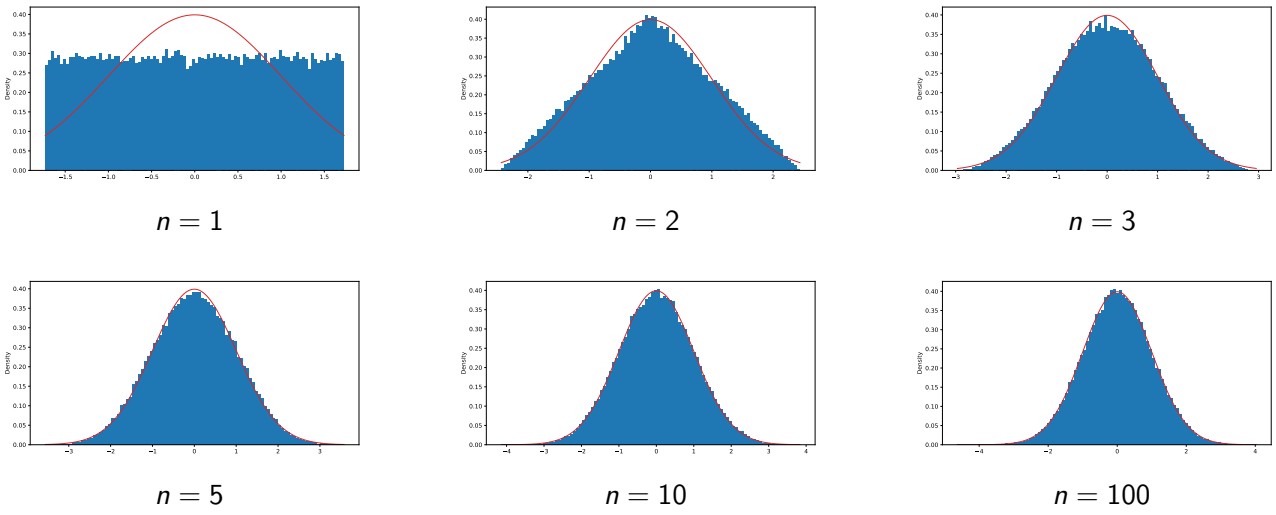


Figure 2.1: Simulated values for Y_n using 10^5 runs for each simulation. As n increases, the histogram resembles the density of a standard normal distribution more and more, illustrating how the CLT works.

Listing 2.1: The source code used for example 2.1.

```
from numpy import std, mean, arange, zeros, sqrt
from scipy import stats
import matplotlib.pyplot as plt
from statsmodels.stats.weightstats import DescrStatsW

Zdist = stats.uniform(0, 1)          # Pick any distribution function here

z = Zdist.mean()                     # 1/2
sigma = Zdist.std()                  # 1/sqrt(12)

n = 10
nrRuns = 100000

Ysim = zeros(nrRuns)
for i in range(nrRuns) :
    Zsim = Zdist.rvs(n)
    Yn = sqrt(n) * (mean(Zsim) - z)/sigma
    Ysim[i] = Yn

plt.figure()
plt.hist(Ysim, bins=100, rwidth=0.8, density=True)
xs = arange(min(Ysim), max(Ysim), 0.01)
ys = stats.norm(0, 1).pdf(xs)
```

If we let z_α be the $1-\alpha$ quantile of the standard normal distribution, that is, $\Phi(z_\alpha) = 1-\alpha$, it follows that

$$\mathbb{P}\left(z_{1-\alpha} < Y_n < z_\alpha\right) \approx 1 - 2\alpha. \tag{2.10}$$

By rewriting the event $\{z_{1-\alpha} < Y_n < z_\alpha\}$, we find that

$$z_{1-\alpha} < \sqrt{n}\frac{\overline{Z} - z}{\sqrt{S^2}} < z_\alpha \tag{2.11}$$

is equivalent to

$$\overline{Z} - z_\alpha\sqrt{\frac{S^2}{n}} < z < \overline{Z} - z_{1-\alpha}\sqrt{\frac{S^2}{n}}, \tag{2.12}$$

so that

$$\mathbb{P}\left(\overline{Z} - z_\alpha\sqrt{\frac{S^2}{n}} < z < \overline{Z} - z_{1-\alpha}\sqrt{\frac{S^2}{n}}\right) \approx 1 - 2\alpha. \tag{2.13}$$

From this, it follows that

$$\left(\overline{Z} - z_\alpha\sqrt{\frac{S^2}{n}}, \overline{Z} - z_{1-\alpha}\sqrt{\frac{S^2}{n}}\right) \tag{2.14}$$

is an approximate $100(1-2\alpha)\%$ confidence interval for z.

We will often use $\alpha = 0.025$. In this case, $z_\alpha = -z_{1-\alpha} \approx 1.96$ and we thus use the 95% confidence interval

$$\left(\overline{Z} - 1.96\sqrt{\frac{S^2}{n}}, \overline{Z} + 1.96\sqrt{\frac{S^2}{n}}\right). \tag{2.15}$$

The quantity $z_\alpha\sqrt{S^2/n}$ is called the *half-width* of the confidence interval. From the factor $\sqrt{n}$ in the denominator, it can be seen that in order to obtain one extra digit of the value z, the number of runs must be increased by a factor 100. To construct a confidence interval for the mean in Python, we can either compute it manually using (2.15), or use the `tconfint_mean` method in `DescrStatsW`. The sample source code is given in Listing 2.2.

Listing 2.2: Computing a confidence interval in Python.

```
from statsmodels.stats.weightstats import DescrStatsW

# the variable sim contains the simulation results
n = len(sim)                    # number of runs
Xavg = mean(sim)                # estimate for E[X]
Sn = std(sim)                   # estimated standard deviation
halfWidth = 1.96 * Sn/sqrt(n)   # half-width of the confidence interval
ci = (Xavg - halfWidth, Xavg + halfWidth) # manually compute confidence interval
print(ci)

# Alternative, using statsmodels
ci = DescrStatsW(sim).tconfint_mean(alpha=0.05)
print(ci)
```

Exercise 2.2. Show that $z_\alpha = -z_{1-\alpha}$.

Solution:

We know that the standard normal distribution is symmetric around zero. It is now clear from Figure 2.2 that $z_{1-\alpha} = -z_\alpha$. Note that the α-*quantile* is commonly defined as $q_\alpha = z_{1-\alpha}$.

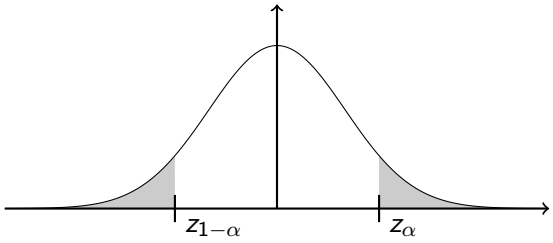


Figure 2.2: α and $1-\alpha$ quantiles of the standard normal distribution. The shaded areas both have area α.

Example 2.2. In order to get some feeling for confidence intervals, consider the simple example where we want to estimate the mean of a uniform random variable on $(-1, 1)$. We will construct 100 approximate 95% confidence intervals, each of which is based on 50 observations. The result can be found in Figure 2.3 and the code used to generate this plot can be found in Listing 2.3. From the plot, it can be seen that most of the constructed interval contain the true expected value (0), while some of the intervals (approximately 5%) do not contain the true value.

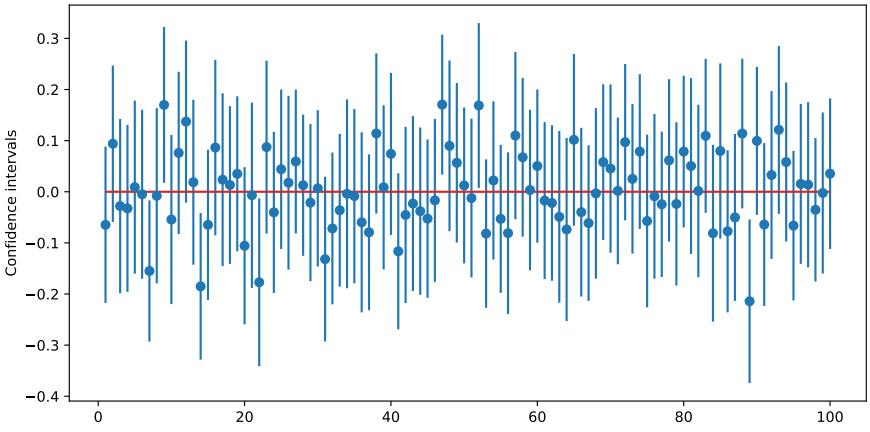


Figure 2.3: 100 confidence intervals for the estimation of the mean of a uniform random variable on $(-1, 1)$. Each interval is based on 50 observations. The red line indicates the theoretical mean.

2.3 Number of runs

The CLT also provides a means for determining the number of runs required to reach the desired level of accuracy of your simulated estimate *before running your simulation*. To do so, you need to have an *idea* of the value of σ.

For example, to obtain a $(1-\alpha)\%$ confidence interval for the mean of Z of half-width $\varepsilon > 0$, the number of replications n has to satisfy the following inequality.

$$z_{\alpha/2} \cdot \frac{\sigma}{\sqrt{n}} < \varepsilon$$

or

$$n > \left(\frac{z_{\alpha/2} \cdot \sigma}{\varepsilon}\right)^2. \tag{2.16}$$

Listing 2.3: The source code used to construct Figure 2.3.

```
# knowing what the true value of E[X] is.
repetitions = 100               # The number of confidence intervals.
observations = 50               # The number of observations per interval.
Zdist = stats.uniform(-1, 2)    # We can use ANY probability distribution here

upper = zeros(repetitions)      # upper limit of confidence intervals.
lower = zeros(repetitions)      # lower limit of confidence intervals.

for run in range(repetitions) :
    r = Zdist.rvs(observations) # Draw a number of uniform random variables.

    sd = std(r)
    m = mean(r)

    halfwidth = 1.96 * sd/sqrt(observations)
    lower[run] = m - halfwidth
    upper[run] = m + halfwidth

plt.figure()
plt.errorbar(x=arange(1, repetitions+1),
             y=(lower + upper) / 2,
             yerr=(upper - lower) / 2,
             fmt='o')
plt.hlines(xmin=1, xmax=repetitions, y=0, color='red')
```

If no initial guess for the value of σ is available, a two-step approach can be used:

1. Perform a short, initial simulation with a relatively small number of runs. Estimate σ from these simulation results.

2. Plug in this estimate in Equation (2.16) and compute how many more runs you need to get the desired accuracy ε.

Special case: Bernoulli random variables. A special case, which does *not* require an initial guess for σ, is when we want to simulate probabilities. When simulating a probability, the outcome of one simulation run is 0 or 1 (see Section 1.1):

$$\mathbb{P}(A) = \mathbb{E}[\mathbf{1}_{\{A\}}].$$

This means that an estimator for the unknown probability p is

$$\widehat{p} := \frac{1}{n}\sum_{i=1}^{n} Z_i,$$

where Z_i has a Bernoulli distribution with this (unknown) parameter p. We have:

$$\mathbb{E}[Z_i] = p, \qquad \mathrm{Var}\,[Z_i] = p(1-p).$$

A $(1-\alpha)$% confidence interval for the parameter p is

$$\widehat{p} \pm \frac{z_{\alpha/2}\sqrt{\widehat{p}(1-\widehat{p})}}{\sqrt{n}}.$$

We can obtain the minimum sample size n from this formula:

$$\left(\frac{z_{\alpha/2}\cdot\sigma}{\varepsilon}\right)^2 = \widehat{p}(1-\widehat{p})\left(\frac{z_{\alpha/2}}{\varepsilon}\right)^2.$$

Although $\widehat{p}$ and, as a consequence, the variance $\sigma^2 = \widehat{p}(1-\widehat{p})$ is unknown, we can use the property that for $0 \leq \widehat{p} \leq 1$ the variance is maximal when $\widehat{p} = \frac{1}{2}$. We can determine the number of runs based on this pessimistic estimate for $\widehat{p}$, which leads to

$$n = \frac{1}{4}\left(\frac{z_{\alpha/2}}{\varepsilon}\right)^2,$$

because $\widehat{p}(1-\widehat{p}) \leq \frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. Note that for $\alpha = 0.05$ we should take (at least) approximately $1/\varepsilon^2$ runs.

Example 2.3. We revisit Example 1.4, where we used stochastic simulation to estimate π. The key observation in that example is that the Bernoulli random variables C_i have expectation $p := \pi/4$. Now define $X_i := 4C_i$, with probability distribution

$$X_i = \begin{cases} 4 & \text{w.p. p,} \\ 0 & \text{w.p. 1 - p,} \end{cases}$$

and remember that p should be considered unknown, as we are trying to estimate π. We find:

$$\sigma^2 = \mathrm{Var}\left[X_i\right] = 4^2\mathrm{Var}\left[C_i\right] = 16p(1-p) \leq 16 \cdot \frac{1}{2} \cdot \frac{1}{2} = 4.$$

Now we can determine the minimum number of runs to estimate π with a given accuracy. If we want to estimate π with k digits accuracy, meaning that $\varepsilon = 10^{-k}$, we should take

$$n = \left(\frac{z_{\alpha/2} \cdot \sigma}{\varepsilon}\right)^2 = \left(\frac{1.96 \cdot 2}{10^{-k}}\right)^2 \approx 16 \cdot 10^k.$$

3

Pseudo-random number generators

A simulation of a system which contains random components requires a method of generating random numbers. This section is devoted to the generation of random numbers from the uniform $(0, 1)$ distribution. Although this is only one of all possible distribution functions, it is very important to have an efficient way of generating uniformly distributed random numbers. This is due to the fact that random variables from other distributions can often be obtained by transforming uniformly distributed random variables. We come back to this issue in the next chapter.

A good random number generator should satisfy the following properties:

1. The generated numbers should satisfy the property of uniformity.
2. The generated numbers should satisfy the property of independence.
3. The random numbers should be replicable.
4. It should take a long time before numbers start to repeat.
5. The routine should be fast.
6. The routine should not require a lot of storage.
7. The method to generate the random numbers should be cryptographically secure.

The generation of random numbers has a long history. In the beginning, random numbers were generated by hand using methods as throwing dice or drawing numbered balls. However, nowadays only numerical ways to generate random numbers are used. Usually, the random numbers are generated in a sequence, each number being completely determined by one or several of its predecessors. Hence, these generators are often called *pseudo-random number generators*.

3.1 Pseudo-Random Number Generators

Most pseudo-random number generators can be characterized by a five-tuple (S, s_0, f, U, g), where S is a finite set of states, $s_0 \in S$ is the initial state, $f : S \to S$ is the transition function, U is a finite set of output values, and $g : S \to U$ is the output function.

The pseudo random-number generator then operates as follows:

- (1) Let $u_0 = g(s_0)$.
- (2) For $i = 1, 2, ...$ let $s_i = f(s_{i-1})$ and $u_i = g(s_i)$.

The sequence $(u_0, u_1, ...)$ is the output of the generator. The choice of a fixed initial state s_0 rather than a probabilistic one makes replication of the random numbers possible.

Let us next give a number of examples of pseudo-random number generators.

3.1.1 Midsquare Method

One of the first pseudo-random number generators was the midsquare method. It starts with a fixed initial number, say of 4 numbers, called the seed. This number is squared and the middle digits of this square become the second number. The middle digits of this second number are then squared again to generate the third number and so on. Finally, we get realizations from the uniform (0,1) distribution after placement of the decimal point (i.e. after division by 10000). The choice of the seed is very important as the next examples will show.

Example 3.1.

- If we take as seed $Z_0 = 1234$, then we will get the sequence of numbers 0.1234, 0.5227, 0.3215, 0.3362, 0.3030, 0.1809, 0.2724, 0.4201, 0.6484, 0.0422, 0.1780, 0.1684, 0.8358, 0.8561, 0.2907, ...
- If we take as seed $Z_0 = 2345$, then we get the sequence of numbers 0.2345, 0.4990, 0.9001, 0.0180, 0.0324, 0.1049, 0.1004, 0.0080, 0.0064, 0.0040, ... Two successive zeros behind the decimal point will never disappear.
- If we choose $Z_0 = 2100$, then we get the sequence 0.2100, 0.4100, 0.8100, 0.6100, 0.2100, 0.4100, ... Only after four numbers the sequence starts to repeat itself.

3.1.2 Linear Congruential Method

Nowadays, most of the random number generators in use are so-called linear congruential generators. They produce a sequence of integers between 0 and $m-1$ according to the following recursion:

$$Z_i = (aZ_{i-1} + c) \bmod m, \qquad i = 1, 2, 3, \ldots .$$

The initial value Z_0 is called the seed, a is called the multiplier, c the increment and m the modulus. To obtain the desired uniformly (0,1) distributed random numbers we should choose $U_i = Z_i/m$. Remark that U_i can be equal to zero, which will cause some problems if we, for example, are going to generate exponentially distributed random variables (see section 4). A good choice of the values a, c and m is very important. One can prove that a linear congruential generator has a maximal cycle length m if the parameters a, c and m satisfy the following properties:

- c and m are relatively prime;
- if q is a prime divisor of m then $a = 1 \bmod q$;
- if 4 is a divisor of m then $a = 1 \bmod 4$.

Example 3.2.

- If we choose $(a, c, m) = (1, 5, 13)$ and take as seed $Z_0 = 1$, then we get the sequence of numbers $1, 6, 11, 3, 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, \ldots$ which has maximal cycle length of 13.
- If we choose $(a, c, m) = (2, 5, 13)$ and take as seed $Z_0 = 1$, then we get the sequence of numbers $1, 7, 6, 4, 0, 5, 2, 9, 10, 12, 3, 11, 1, \ldots$ which has cycle length of 12. If we choose $Z_0 = 8$, we get the sequence $8, 8, 8, \ldots$. (cycle length 1!)

3.1.3 Additive Congruential Method

The additive congruential method produces a sequence of integers between 0 and $m-1$ according to the recursion

$$Z_i = (Z_{i-1} + Z_{i-k}) \bmod m, \qquad , i = 1, 2, 3, \dots ,$$

where $k \geq 2$. To obtain the desired uniformly (0,1) distributed random numbers we again should choose $U_i = Z_i/m$. The method can have a maximal cycle length of m^k. However, the method has also some disadvantages. Consider for example the special case $k = 2$. Now, if we take three consecutive numbers U_{i-2}, U_{i-1} and U_i, it will never happen that $U_{i-2} < U_i < U_{i-1}$ or $U_{i-1} < U_i < U_{i-2}$. (For three independent, uniformly (0,1) distributed random variables both of these orderings have probability 1/6).

3.1.4 Tausworthe Generators

The recursions in the linear congruential method and in the additive congruential are special cases of the recursion

$$Z_i = (\sum_{j=1}^{k} a_j Z_{i-j} + c) \bmod m, \qquad i = 1, 2, 3, \dots .$$

A special situation is obtained if we take $m = 2$ and all $a_j \in \{0, 1\}$. In that case the generator produces a sequence of bits. Such a generator is often called a *Tausworthe generator*, or a *shift register generator*. The sequence $Z_0, Z_1, Z_2, \dots$, is now transformed into a sequence $U_0, U_1, U_2, \dots$, of uniform random numbers in the following way: Choose an integer $l \leq k$ and put

$$U_n = \sum_{j=0}^{l-1} Z_{nl+j} 2^{-(j+1)}.$$

In other words, the numbers U_n are obtained by splitting up the sequence $Z_0, Z_1, Z_2, \dots$ into consecutive blocks of length l and then interpreting each block as the digit expansion in base 2 of a number in [0,1).

In practice, often only two of the a_j's are chosen equal to one, so we get

$$Z_i = (Z_{i-h} + Z_{i-k}) \bmod 2.$$

Example 3.3. If we choose $h = 3$ and $k = 4$ and start with the initial values 1,1,0,1, then we get the following sequence of bits $1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, \ 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, \dots$. The sequence is periodic with period $2^k - 1 = 15$. If we take $l = 4$, this leads to the random numbers $13/16$, $7/16$, $8/16$, $9/16$, $10/16$, $15/16$, $1/16$, $3/16$, ...

3.1.5 State-of-the-art

Java still uses a Linear Congruential Generator with a 48-bit seed, meaning that $m = 2^{48}$. The other parameters are chosen such that the generator has maximum period. However, more sophisticated methods have been developed in the recent literature. The “best” pseudo-random number generator at this moment appears to be the Mersenne Twister. Its name is derived from the fact that its period length is chosen to be a Mersenne prime. The most commonly-used version of the Mersenne Twister algorithm is based on the Mersenne prime $2^{19937} - 1$. It is the standard random number generator in most software packages, including Python and R. Although it is not the standard pseudo-random number generator in Java, a Java implementation is also available. More information can be found in [MN98].

3.1.6 Cryptographically secure pseudo-random number generators (CSPRNGs)

One of the desirable properties of pseudo-random number generators is that the method used generate the random numbers is cryptographically secure. Although this property is not directly relevant for the applications that are discussed in the context of this Stochastic Simulation course, it is extremely important in practical applications that involve generation of random cryptographic keys. In this context, desirable properties are:

- given only a number produced by the generator, it is impossible to predict previous and future numbers;
- the numbers produced contain no known biases;
- the generator has a large period;
- the generator can seed itself at any position within that period with equal probability.

For example, when using the generator to produce a session ID on a web server, we do not want user n to predict user $(n+1)$'s session ID.

Although the standard built-in random number generators are not cryptographically secure, most programming languages have secure alternatives. For example, in Python, you can use `os.urandom()`, in R you can use the `randaes` package, and in Java you can use use `java.security.SecureRandom`. Unfortunately, CSPRNGs are *much* slower than non-secure alternatives.

3.1.7 Random seeds

As discussed earlier in this chapter, all random number generators are initialised with a *seed*. Given the same initial seed, the random number generator will produce the same list of pseudo random numbers, every time. In order to produce an *unpredictable* (random) list of pseudo-random numbers, one would like to use a random seed, which sounds like a typical "chicken or egg" problem. In earlier days, this was solved by using "something random" like the current time at which the random number generator was created. This created unsafe situations, however, where it might be easy to guess the random seed or where unwanted correlation between different (sequences of) random numbers was created. For more information, see also the Bitrandomness example in Section 3.4). Fortunately, this problem is solved on modern computers by the operating system and/or a hardware device that generates the random numbers.

In this subsection we discuss random seeds in more detail, because it is sometimes desirable to be able to reproduce the same sequences of pseudo-random numbers. For this reason, we explain how to appropriately use random number generators in Python, or any other programming language. Python, in particular, has improved its random number generation implementation considerably since approximately 2020. The preferred way to sample random numbers is to first create a random number generator object, which is then used in all other objects or functions that generate random numbers. See Listing 3.1 for an example in Python.

Listing 3.1: Specifying random seeds in Python.

```
1  from scipy import stats
2  from numpy.random import default_rng
3
4  rng = default_rng(1234)        # Specify a random seed
5
6  normDist1 = stats.norm(0, 1)   # Construct the probability distributions
7  normDist2 = stats.norm(0, 1)
8  poissonDist = stats.poisson(10)
9
10 # Specify, for each distribution that you create, this random number generator:
11 normDist1.random_state = rng
```

```
12  normDist2.random_state = rng
13  poissonDist.random_state = rng
14
15  x1 = normDist1.rvs(10)     # Sample the random numbers.
16  x2 = normDist2.rvs(10)     # The same random numbers will be generated
17  y = poissonDist.rvs(10)    # every time you run this program.
```

In some situations, one might need to have the guarantee that all the random samples for one or more specific variables are the same. This situation arises, for example, in more complex discrete event systems as discussed in Chapter 8 with multiple random processes taking place simultaneously. To ensure that in a queueing system the service times and the interarrival times of customers are reproducible, it might be necessary to use a separate random number generator for each random variable.

3.2 Tests of Random Number Generators

The main properties that a random number generator should have are uniformity and independence. In this section we will describe two tests, the Kolmogorov-Smirnov test and the chi-square test, to compare the distribution of the set of generated numbers with a uniform distribution. Furthermore, we will describe a number of tests that are able to check whether or not the set of generated numbers satisfies independence.

As we have seen before, R uses a more sophisticated pseudo-random number generator than Java (February 2014). The effect can be illustrated very nicely in the following example.

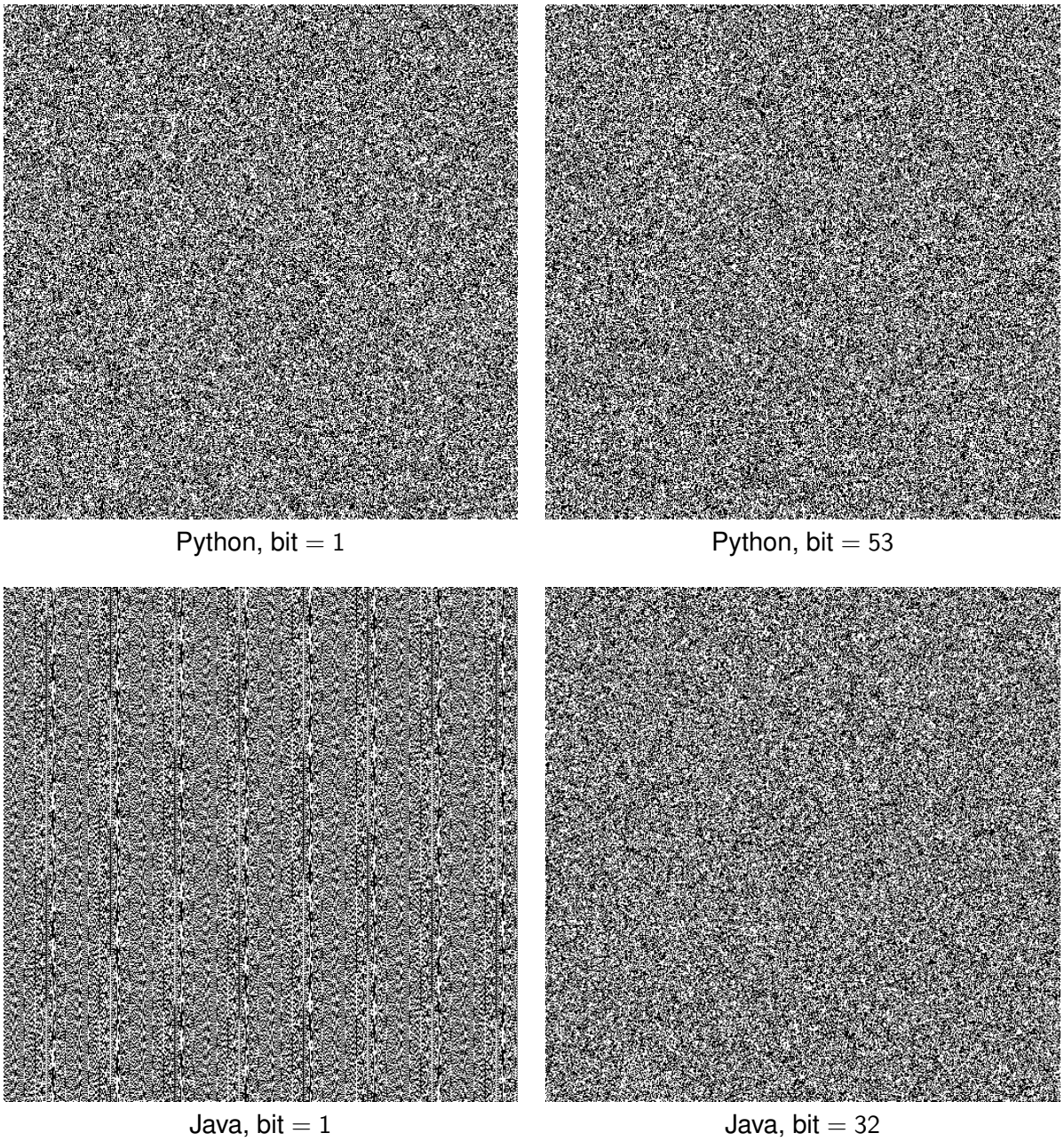
Example 3.4. The random numbers generated by LCGs have the following undesirable property: randomness depends on the bit position! The Mersenne Twister used by most statistical packages, for example R and Python, does not seem to have this issue. In this example we will generate 512×512 random integers between and convert these random integers to booleans by checking whether the first bit equals 0 or 1. Subsequently, we repeat this experiment but study the last bit (which is the 32nd bit in Java and R, and the 53rd bit in Python). The Python code used in this example is printed in listing 3.2. The results for Python and Java are plotted in Figure 3.1 (the results for R are comparable to Python). As can be seen clearly, the Java random number generator exhibits a pattern when using the first bit to generate random booleans. For this reason, the function `nextBoolean()` should be used instead of checking the first bit, which corresponds to a check whether the integer is odd or even.

3.2.1 The Kolmogorov-Smirnov test

This test compares the empirical distribution $F_N(x)$ of a set of N generated random numbers with the distribution function $F(x) = x, 0 \leq x \leq 1$, of a uniformly distributed random variable. Here, $F_N(x)$ is defined as the number of observations smaller than or equal to x divided by N, the total number of observations. Let α be the significance level of the test, i.e. α is the probability of rejecting the null hypothesis that the numbers are uniformly distributed on the interval $(0, 1)$ given that the null hypothesis is true. Under the null hypothesis $F_N(x)$ will tend to $F(x)$ as N tends to infinity. The Kolmogorov-Smirnov test is based on

$$D = \max | F(x) - F_N(x) | ,$$

the maximal absolute difference between $F_N(x)$ and $F(x)$ over the range of the random variable. Now, if the value of D is greater than some critical value D_α the null hypothesis is rejected. If $D \leq D_\alpha$, we conclude that no difference has been detected between the empirical distribution of the generated numbers and the uniform distribution. The critical value D_α for the specified significance level α of the test and the given sample size N can be found in a table on the Kolmogorov-Smirnov test. A fast Python implementation is given in Listing 3.3.



Python, bit $= 1$ Python, bit $= 53$

Java, bit $= 1$ Java, bit $= 32$

Figure 3.1: Images generated from random integers and taking the first and the last bit, respectively. It is clear that the Java random number generator has issues with independence of the first bit.

Listing 3.2: Generating random booleans by taking the first bit of random integers.

```
from numpy import zeros, floor
from scipy import stats
import matplotlib.pyplot as plt
import matplotlib.cm as cm

# Bit randomness
# Python uses the Mersenne Twister as the core generator.
# It produces 53-bit precision floats

def generatePattern(n, bitNr):
    b = zeros(n * n)
    nn = 53  # nr of bits of Python's rng
    rdist = stats.uniform(0, 2**nn)
    r = floor(rdist.rvs(n*n))
    for i in range(n * n):
        ri = int(r[i])
        bit = (ri & 1 << (bitNr - 1))
        b[i] = bit > 0
    bArray = b.reshape(n, n)
    return bArray

plt.figure()
plt.imshow(generatePattern(512, 1), cmap=cm.gray)
plt.figure()
plt.imshow(generatePattern(512, 53), cmap=cm.gray)
plt.show()
```

Listing 3.3: distribution.]The Kolmogorov-Smirnov test in Python, used to check whether the data in the vector *r* fits a uniform [0, 1] distribution.

```
n = 1000
Zdist = stats.uniform(0, 1)         # sample 1000 random U[0, 1] numbers
r = Zdist.rvs(n)

tst = stats.kstest(r, Zdist.cdf)    # Kolmogorov-Smirnov test
print(tst)                          # If p-value < alpha, reject H_0
```

3.2.2 The chi-square test

In the chi-square test, we divide the interval $(0, 1)$ in n subintervals of equal length. The test uses the statistic

$$Y = \sum_{i=1}^{n} \frac{(Y_i - E(Y_i))^2}{E(Y_i)},$$

where Y_i is the number of realizations in the i-th subinterval. Clearly, if N is the total number of observations, we have $E(Y_i) = N/n$. Under the null hypothesis of uniformity of the realizations, it can be shown that the distribution of Y is approximately the chi-square distribution with $n - 1$ degrees of freedom. So, we compare the value of our statistic Y with the α-percentile of the chi-square distribution with $n - 1$ degrees of freedom to conclude whether or not we reject the null hypothesis. Here, α is again the significance level of the test. It is recommended to choose n and N such that $N \geq 50$ and $E(Y_i) \geq 5$.

3.2.3 The serial test

The serial test is a 2-dimensional version of the uniformity test of the previous subsection to test the independence between successive observations. Therefore, we look at N successive tuples $(U_1, U_2), (U_3, U_4), \ldots, (U_{2N-1}, U_{2N})$ of our observations and we count how many observations fall into the N^2 different subsquares of the unit square. We then apply a chi-square test to these data. Of course, we can also formulate a higher-dimension version of this test.

3.2.4 The permutation test

In the permutation test we look at N successive k-tuples of realizations $(U_0, \dots, U_{k-1})$, $(U_k, \dots, U_{2k-1})$, ..., $(U_{(N-1)k}, \dots, U_{Nk-1})$. Among the elements in a k-tuple there are $k!$ possible orderings and these orderings are all equally likely. Hence, we can determine the frequencies of the different orderings among the N different k-tupels and apply a chi-squared test to these data.

3.2.5 The run test

In the run test we divide the realizations in blocks, where each block consists of a sequence of increasing numbers followed by a number which is smaller than its predecessor. For example, if the realizations are 1,3,8,6,2,0,7,9,5, then we divide them in the blocks (1,3,8,6), (2,0) and (7,9,5). A block consisting of $k+1$ numbers is called a *run-up of length* k. Under the null hypothesis that we are dealing with independent, uniformly distributed random variables the probability of having a run-up of length k equals $1/k! - 1/(k+1)!$. In a practical implementation of the run test we now observe a large number N of blocks. Furthermore, we choose an integer h and count the number of runs-up of length $0, 1, 2, \dots, h-1$ and $\geq h$. Then we apply a chi-square test to these data. Of course, we can do a similar test for runs-down.

3.2.6 The gap test

Let J be some fixed subinterval of $(0, 1)$. If we have that $U_{n+j} \notin J$ for $0 \leq j \leq k$ and both $U_{n-1} \in J$ and $U_{n+k+1} \in J$, we say that we have a *gap of length* k. Under the null hypothesis that the random numbers U_n are independent and uniformly distributed on (0,1), we have that the gap lengths are geometrically distributed with parameter p, where p is the length of interval J (i.e. P(gap of length k) = $p(1-p)^k$). In a practical implementation of the gap test we observe again a large number N of gaps. Furthermore, we choose an integer h and count the number of gaps of length $0, 1, 2, \dots, h-1$ and $\geq h$. Then we apply a chi-square test to these data.

3.2.7 The serial correlation test

In the serial correlation test we calculate the serial correlation coefficient

$$R = \sum_{j=1}^{N} (U_j - \overline{U})(U_{j+1} - \overline{U}) / \sum_{j=1}^{N} (U_j - \overline{U})^2,$$

where $\overline{U} = \sum_{j=1}^{N} U_j/N$ and U_{N+1} should be replaced by U_1. If the U_j's are really independent the serial correlation coefficient should be close to zero. Hence we reject the null hypothesis that the U_j's are independent if R is too large. The exact distribution of R is unknown, but for large values of N we have that, if the U_j's are independent, then $P(-2/\sqrt{N} \leq R \leq 2/\sqrt{N}) \approx 0.95$. Hence, we reject the hypothesis of independence at the 5% level if $R \notin (-2/\sqrt{N}, 2/\sqrt{N})$.

4

Sampling random variables

Python has many built-in functions to generate random variables according to a certain distribution. We prefer the `scipy.stats` module, because it has a nice object oriented structure, defining generic classes for probability distributions. A few examples are listed in Table 4.1. Each of these probability distributions has a method `rvs(n)` that samples n random numbers from this distribution. However extensive this list may be, it does only cover the distributions that are most common. In general, you may want to draw random variables from another distribution. It may also be the case that you have to implement simulations in a different programming language, for which such a large library of functions does not exist.

There are many algorithms to draw random variables from different distributions. The aim of this chapter is to provide a few general methods, that work in many cases. It should be noted, however, that these methods are not always efficient.

We will assume that we have a method to draw Uniform $(0, 1)$ random variables available. All programming languages provide a function to generate such random variables. Given a method to draw Uniform $(0, 1)$ random variables, we can easily draw from a Uniform (a, b) distribution: if U follows a uniform distribution on $(0, 1)$, it follows that $a + (b - a)U$ follows a uniform distribution on (a, b).

Distribution	**Function**
Binomial	`binom(n, p)`
Poisson	`poisson(lambda)`
Geometric (on $1, 2, ...$)	`geom(p)`
Hypergeometric	`hypergeom( m, n, k)`
Negative binomial	`nbinom(n, p)`
Exponential	`expon(scale=1/lambda)`
Normal	`norm(mu, sigma)`
Gamma	`gamma(alpha, scale=1/beta)`
Uniform	`uniform(a, b - a)`

Table 4.1: List of (some) probability distributions in Python.

Remark: By default, probability distributions in Python's `scipy.stats` module all have a *location* and *scale* parameter. Although this consistency certainly has some advantages, it also implies that the parametrisation may be different than what is common in the existing literature. Please check Table 4.1 and the examples in these lecture notes carefully to make sure that you use the right parameters. In case of doubt, read the Python documentation.

4.1 Discrete random variables

Suppose that U is uniform on $(0,1)$ and that $p \in (0,1)$. The probability that $U \leq p$ is then just p. So if we want to draw a Bernoulli random variable, X say, that attains the value 1 with probability p, we can just take U and set $X = 1$ if $U \leq p$ and set $X = 0$ otherwise.

This observation can be extended to more general discrete random variables. Assume that we want to create a random variable that takes on values $x_1, x_2, \ldots x_n$ with probability $p_1, p_2, \ldots, p_n$, respectively. This can be done as follows: take a uniform random variable on $(0,1)$ and set $X = x_1$ if $U \leq p_1$, $X = x_2$ if $p_1 < U \leq p_1 + p_2$ and so on. It follows easily that $\mathbb{P}(X = x_i) = p_i$, for $i = 1, 2, \ldots, n$.

Exercise 4.1. Write a function that generates a realisation of a $\text{Bin}(n,p)$ random variable (without using built-in function to perform this action, such as `binom` and `sample`).

Solution: Sample code can be found in Listing 4.1.

Listing 4.1: A function for drawing binomial random variables.

```
1  def sampleBinom(nrSamples, n, p):
2      probs = [binom(n, k) * p**k * (1 - p)**(n - k) for k in range(n + 1)]
3      cumProbs = cumsum(probs)
4      u = stats.uniform(0, 1).rvs(nrSamples)
5      k = [sum(cumProbs < x) for x in u]
6      return k
```

Obviously, the implementation in Listing 4.1 would have to be adapted when applying this method to a random variable with infinite support, because we cannot create a finite-size array `probs` containing all the probabilities. This method is called the *Discrete version of the Inverse transformation method*. Its continuous counterpart is discussed in Section 4.2.

In case the number of possible values $x_1, \ldots, x_n$ is finite and not excessively large, a more efficient method is available, called the Array method. As an example, suppose $p_i = k_i/100$, for $i = 1, \ldots, n$, where the k_i's are integers with $0 \leq k_i \leq 100$. We can now construct a list (or array) A as follows:

$$A[i-1] = \begin{cases} x_1 & \text{for } i = 1, \ldots, k_1, \\ x_2 & \text{for } i = k_1 + 1, \ldots, k_1 + k_2, \\ \vdots & \\ x_n & \text{for } i = k_1 + \cdots + k_{n-1} + 1, \ldots, k_1 + \cdots + k_n. \end{cases}$$

Then, first sample a random index I from $0, \ldots, 99$, for example using $I = \lfloor 100U \rfloor$, and set $X = A[I]$. This can be generalised to any random variable where the possible outcomes can be written as a quotient of two integers. If this is not the case, one can choose to round the outcomes to a chosen number of digits. See Listing 4.2 for sample code on how to use the array method to sample from a random variable X with

$$P(X = 1) = 0.2, P(X = 2) = 0.13, P(X = 3) = 0.52, P(X = 4) = 0.25.$$

Listing 4.2: Array method for sampling discrete random variables.

```
1  A = [1] * 20 + [2] * 13 + [3] * 52 + [4] * 25
2  unif = stats.uniform(0, 1)
3  I = int(floor(unif.rvs() * 100))
4  print(I, A[I])
```

Although the inverse transformation method can be used for the generation of arbitrary discrete random variables, it is not always the most efficient method. Hence we shall give some alternative methods for some special distributions.

4.1.1 The geometric distribution

A random variable X has a geometric distribution with parameter p if

$$P(X = k) = p(1-p)^{k-1}, \quad k = 1, 2, 3, ...$$

The geometric distribution can be interpreted as the distribution of the waiting time until the first head in a sequence of independent coin tossing experiments, where p equals the probability of throwing a head. Hence, if $U_1, U_2, ...$ are independent, identically distributed uniform (0,1) random variables and X is the index of the first U_i for which $U_i \leq p$, then X is geometrically distributed with parameter p.

Another way to generate a geometric random variable is by using the fact that if U is uniform (0,1), then

$$\left\lceil \frac{\ln(U)}{\ln(1-p)} \right\rceil$$

is geometric with parameter p.

4.1.2 The binomial distribution

A random variable X has a binomial distribution with parameters n and p if

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, 2, ..., n.$$

The binomial distribution can be interpreted as the distribution of the number of heads in a sequence of n independent coin tossing experiments, where p equals the probability of throwing a head. Hence, if $U_1, ..., U_n$ are independent, identically distributed uniform (0,1) random variables, then

$$X = \sum_{i=1}^{n} 1_{[U_i \leq p]}$$

is binomial distributed with parameters n and p. An alternative generation of a binomial distributed random variable can be obtained using one of the following two lemma's.

Lemma 4.1. *Let $G_1, G_2, ...$ be independent, identically distributed geometric random variables with parameter p. If X is the smallest integer such that*

$$\sum_{i=1}^{X+1} G_i > n,$$

then X is binomial distributed with parameters n and p.

Lemma 4.2. *Let $E_1, E_2, ...$ be independent, identically distributed exponential random variables with parameter 1. If X is the smallest integer such that*

$$\sum_{i=1}^{X+1} \frac{E_i}{n-i+1} > -\ln(1-p),$$

then X is binomial distributed with parameters n and p.

4.1.3 The Poisson distribution

A random variable X has a Poisson distribution with parameter λ if

$$P(X = k) = \frac{\lambda^k}{k!} e^{-\lambda}, \quad k = 0, 1, 2, ...$$

An alternative generation of a Poisson distributed random variable can be obtained using one of the following two lemma's.

Lemma 4.3. *Let $E_1, E_2, \ldots$ be independent, identically distributed exponential random variables with parameter 1. If X is the smallest integer such that*

$$\sum_{i=1}^{X+1} E_i > \lambda,$$

then X is Poisson distributed with parameter λ.

Lemma 4.4. *Let $U_1, U_2, \ldots$ be independent, identically distributed uniform (0,1) random variables. If X is the smallest integer such that*

$$\prod_{i=1}^{X+1} U_i < e^{-\lambda},$$

then X is Poisson distributed with parameter λ.

4.2 Inverse transformation method

The *inverse transformation method* is widely applied for simulating continuous random variables that is often used. Its name stems from the fact that the inverse of the cdf is used in the construction of a random variable with the desired distribution. The method can be explained best by an example.

Exercise 4.2 (T)**.** If U follows a uniform distribution on $(0, 1)$, show that

$$X := -1/\lambda \log(1 - U) \tag{4.1}$$

follows an exponential distribution with parameter λ. Also show that $Y := -1/\lambda \log(U)$ follows an exponential distribution with parameter λ.

Solution: We need to show that the cdf of X has the form $1 - \exp(-\lambda x)$. This can be shown as follows:

$$\begin{aligned}\mathbb{P}(X \leq x) &= \mathbb{P}(-1/\lambda \log(1 - U) \leq x) = \mathbb{P}(\log(1 - U) \geq -\lambda x) \\ &= \mathbb{P}(1 - U \geq \exp(-\lambda x)) = \mathbb{P}(U \leq 1 - \exp(-\lambda x)) = 1 - \exp(-\lambda x). \end{aligned} \tag{4.2}$$

The second assertion follows immediately from the facts that U and $1 - U$ are both Uniform $(0, 1)$ distributed random variables.

The approach used in the above exercise is applicable more generally. First, we define the quantile function F^{-1} of a distribution function F. If $u \in [0, 1]$, we set

$$F^{-1}(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}. \tag{4.3}$$

We are now able to show that

Theorem 4.1. *If F is a cdf and U is a uniform random variable on $(0, 1)$, then $F^{-1}(U)$ is distributed according to F.*

Proof. Let $U \sim$ Uniform $(0, 1)$ and let F^{-1} be the inverse of a cdf, as defined in Equation (4.3). Note that F is non-decreasing, from which it follows that $F^{-1}(u) \leq x$ if and only if $\inf\{y : F(y) \geq u\} \leq x$ (by definition of F^{-1}) if and only if $u \leq F(x)$. From this, it follows that

$$\mathbb{P}(F^{-1}(U) \leq x) = \mathbb{P}(U \leq F(x)) = F(x), \tag{4.4}$$

which shows that F is the cdf of the random variable $F^{-1}(U)$. □

Using this theorem, we are now able to generate random variables with cdf F, *provided that we know the quantile function*.

Exercise 4.3 (T)**.** The CDF of the Pareto distribution is given by

$$F(x) = \begin{cases} 0 & \text{if } x < x_m \\ 1 - \left(\frac{x_m}{x}\right)^{\alpha} & \text{if } x \geq x_m \end{cases} \tag{4.5}$$

where x_m is called the location parameter and $\alpha > 0$ is called the shape parameter. Given a random variable U, that is uniformly distributed on $(0, 1)$, construct a random variable that is distributed according to a Pareto distribution with parameters x_m and α.

Solution: Let $u > 0$. Note that $F(x) = u$ if and only if $1 - (x_m/x)^{\alpha} = u$, from which it follows that $x_m/x = (1-u)^{1/\alpha}$, and hence $x = x_m(1-u)^{-1/\alpha}$. From this, it follows that the random variable X, defined by

$$X = \frac{x_m}{(1-U)^{\frac{1}{\alpha}}}, \tag{4.6}$$

follows a Pareto distribution with parameters x_m and α. The same holds for the random variable $Y = x_m(U)^{-1/\alpha}$, since U and $1 - U$ follow the same distribution.

Exercise 4.4. Often, the inverse F^{-1} cannot be written down explicitly. In this case, numerical inversion can be used. Write a Python function that is able to compute the numerical inverse of the cdf of a gamma distribution. Hint: take a look at the function `root` from the `scipy.optimize` module, or write a numerical routine yourself. Note that numerical inversion is relatively slow, in particular considering that one generally needs a very large number of samples. The next section describes a preferred method, that also works for any probability distribution.

Solution: For simplicity, we will assume that $F : [0, \infty) \to [0, 1]$ is an invertible function. Suppose that we want to calculate the value $x = F^{-1}(y)$ for some $y \in [0, 1]$. This is the same as requiring that $F(x) = y$, or equivalently, $F(x) - y = 0$. So if y is some fixed value, we can find x by solving $F(x) - y = 0$ for x. This is exactly what the function `root` does.

The function `root`, in its basic form, takes two parameters: a function and an initial guess (i.e., starting point for the algorithm) for a root of the specified function. Suppose that we have a function called `f`, and we want to solve $f(x) = 0$, and we think that x must lie somewhere near 1. We then use `root(f, 1)` to find the value of x. To improve the accuracy of the solution, increase the tolerance by specifying (for example) `tol=10^-10` to get a ten-digit accuracy.
For an example, see Listing 4.3. Note that `root` returns a lot of information, not only the value of x. To retrieve just the value of x, you can use `root.x`.

We are now ready to use the function `root` for the numerical inversion of the function F. For an example of how to use this approach to sample from a gamma distribution, see the source code in Listing 4.4.

4.3 Acceptance-Rejection Method

Suppose that we want to generate a random variable X with distribution function F and probability density function f. The acceptance-rejection method requires a function g which majorises the density f, i.e. $g(x) \geq f(x)$ for all x. Clearly, $g(x)$ will not be a probability density function because $c := \int g(x)dx > 1$. However, if $c < \infty$, then $h(x) = g(x)/c$ is a density. The method applies if we

Listing 4.3: Sample source code for using the function `root`. We use this function to calculate the square root of 2.

```
from scipy.optimize import root

def f(x):
    return(x**2 - 2)

r = root(f, 1, tol=10**-10)
print(r.x)

# Note that in case of multiple roots,
# the solution depends on the initial guess
r = root(f, -1, tol=10**-10)
print(r.x)
```

Listing 4.4: Source code for computing the numerical inverse of the cdf of a gamma distribution.

```
def sampleGamma(alpha, beta):
    gammaDist = stats.gamma(alpha, scale=1/beta)
    unifDist = stats.uniform(0, 1)
    u = unifDist.rvs()
    def f(x):
        return gammaDist.cdf(x) - u
    y = root(f, alpha/beta)
    return y.x

alpha = 6.25
beta = 2.5
n = 10000
x = [sampleGamma(alpha, beta) for _ in range(n)]
print(mean(x))
print(var(x))
```

are able to generate easily a random variable Y with density h. The method consists of three steps. First, we generate Y having density h. Next, we generate a uniformly distributed random variable U on the interval $(0, 1)$. Finally, we set $X = Y$ if $U \leq f(Y)/g(Y)$ or, if not, we go back to the first step.

Theorem 4.2. *The random variable X generated by the acceptance-rejection method has probability density function f. The number of iterations of the algorithm that are needed is a geometric random variable with mean c.*

Proof: The probability that a single iteration produces an accepted value which is smaller than x equals

$$P(Y \leq x, Y \text{ is accepted}) = \int_{-\infty}^{x} \frac{f(y)}{g(y)} h(y)dy = \frac{1}{c} \int_{-\infty}^{x} f(y)dy$$

and hence each iteration is accepted with probability $1/c$. As each iteration is independent, we see that the number of iterations needed is geometric with mean c. Furthermore,

$$P(X \leq x) = \sum_{n=1}^{\infty} (1 - 1/c)^{n-1} 1/c \int_{-\infty}^{x} f(y)dy = \int_{-\infty}^{x} f(y)dy.$$

Example 4.1 (Generating Gamma distributed random variables)**.** The density of the Gamma distribution is given by

$$f(t) = \frac{1}{\Gamma(\alpha)} \mathrm{e}^{-\beta t} \beta^{\alpha} t^{\alpha-1}, \qquad t \geq 0,$$

where $\Gamma(\alpha) = \int_0^{\infty} \mathrm{e}^{-t} t^{\alpha-1} \mathrm{d}t$. The positive parameters α and β are respectively the shape and rate parameter. Sampling from a Gamma distribution can be done by applying the Acceptance-Rejection method in a smart way. The first step is to use the following properties from a gamma distribution:

- If X is Gamma distributed with parameters α and β, then $Y := X/\beta$ is Gamma distributed with parameters α and 1, which has the simplified density

$$f_0(t) = \frac{1}{\Gamma(\alpha)} \mathrm{e}^{-t} t^{\alpha-1}, \qquad t \geq 0.$$

- If X_1 and X_2 are two independent Gamma distributed random variables with parameters $(\alpha_1, 1)$ and $(\alpha_2, 1)$ respectively, then $X_1 + X_2$ is Gamma distributed with parameter $(\alpha_1 + \alpha_2, 1)$.
- The case where α is an integer, corresponds to an Erlang distribution with α phases.

Using these three properties, the problem of sampling from a Gamma distribution with parameters (α, β) is reduced to sampling from an Erlang distribution with $\lfloor \alpha \rfloor$ phases (which reduces to sampling from an exponential distribution, see Section 4.5), and sampling from a Gamma distribution with parameters $(a, 1)$ where $0 < a < 1$. We will use the Acceptance-Rejection method for this last step, noting that for $0 < a < 1$ the function

$$\mathrm{e}^{-t} t^{a-1} = \frac{1}{\mathrm{e}^t \, t^{1-a}} \leq \begin{cases} t^{a-1} & \text{if } t \leq 1, \\ \mathrm{e}^{-t} & \text{if } t > 1. \end{cases}$$

This suggests using the majorisation function

$$g(t) := \begin{cases} \frac{1}{\Gamma(a)} t^{a-1} & \text{if } t \leq 1, \\ \frac{1}{\Gamma(a)} \mathrm{e}^{-t} & \text{if } t > 1. \end{cases}$$

Normalisation of this function leads to $h(t) := g(t)/c$ with

$$c = \int_0^\infty g(t)\mathrm{d}t = \frac{1}{\Gamma(a)} \left[\int_0^1 t^{a-1} \mathrm{d}t + \int_1^\infty \mathrm{e}^{-t} \mathrm{d}t \right] = \frac{1}{\Gamma(a)} \left(\frac{1}{a} + \frac{1}{\mathrm{e}} \right)$$

Now all we need to do, is sample from a random variable Y having density $h(t)$. This is not too difficult, since both parts have invertible integrals, which means that we can use the inverse transformation method. With probability $\mathrm{e}/(\mathrm{e} + a)$ we sample from $h(t)$ with $t \leq 1$, and with probability $a/(\mathrm{e} + a)$ we sample from the second part of $h(t)$ corresponding to $t > 1$. First we determine the cdf $H(t)$ of Y:

$$H(t) = \int_0^t h(x)\mathrm{d}x = \begin{cases} \left(\frac{1}{a} + \frac{1}{\mathrm{e}} \right)^{-1} \int_0^t x^{a-1} \mathrm{d}x = \frac{\mathrm{e}}{\mathrm{e}+a} t^a & \text{if } t \leq 1, \\ \frac{\mathrm{e}}{\mathrm{e}+a} + \left(\frac{1}{a} + \frac{1}{\mathrm{e}} \right)^{-1} \int_1^t \mathrm{e}^{-x} \mathrm{d}x = 1 - \frac{a\mathrm{e}}{a+\mathrm{e}} e^{-t} & \text{if } t > 1, \end{cases}$$

Now we determine the inverse functions for both parts of $H(t)$:

$$H^{-1}(u) = \begin{cases} \left(\frac{\mathrm{e}+a}{\mathrm{e}} u \right)^{1/a} & \text{if } u \leq \frac{\mathrm{e}}{\mathrm{e}+a}, \\ -\log\left(\frac{\mathrm{e}+a}{a\mathrm{e}} (1-u) \right) & \text{if } u > \frac{\mathrm{e}}{\mathrm{e}+a}. \end{cases}$$

Now we are ready to formulate the complete algorithm to sample from a Gamma distribution. First we sample u from a $U[0,1]$ distribution. Let $b := \frac{\mathrm{e}+a}{\mathrm{e}}$ and $p := ub$. If $p \leq 1$ we return $p^{1/a}$, else we return $-\log((b-p)/a)$. We use the acceptance-rejection method to determine whether our sampled random variate is accepted. The complete implementation can be found in Listing 4.5. Note that in the literature variations of this algorithm have been developed that are slightly more efficient.

Listing 4.5: Sampling from a Gamma distribution.

```
def sampleGamma1(alpha, beta):
    unif = stats.uniform(0, 1)
    k = int(floor(alpha))
    a = alpha - k
    y = 0
    e = exp(1)
    if a > 0 :
        while y == 0:
            u = unif.rvs()
            b = (e + a) / e
            p = u * b
            if p <= 1:
                x = p ** (1/a)
                u1 = unif.rvs()
                if u1 <= exp(-x) :  # Accept
                    y = x
            else :
                x = -log((b - p)/a)
                u1 = unif.rvs()
                if u1 <= x ** (a-1) : # Accept
                    y = x
    product = prod(unif.rvs(k))
    return (y - log(product))/beta

def sampleGamma(nrSamples, alpha, beta):
    return [sampleGamma1(alpha, beta) for _ in range(nrSamples)]
```

4.4 Composition Method

The composition method applies when the distribution function F from which we want to sample can be expressed as a convex combination of other distribution functions $F_1, F_2, ...$, i.e.

$$F(x) = \sum_{j=1}^{\infty} p_j F_j(x)$$

with $p_j \geq 0$ for all j and $\sum_{j=1}^{\infty} p_j = 1$. The method is useful when it is more easy to generate random variables with distribution function F_j then to generate a random variable with distribution function F. The method consists of two steps. First, we generate a discrete random variable J, such that $P(J = j) = p_j$. Next, given $J = j$, we generate a random variable X with distribution function F_j. It is easily seen that the random variable X has the desired distribution function F. The method can be used, for example, to generate a hyperexponentially distributed random variable.

4.5 Convolution Method

For some important distributions, the desired random variable X can be expressed as the sum of other independent random variables, i.e. $X = Y_1 + \cdots + Y_n$, which can be generated more easily than the random variable X itself. The convolution method simply says that you first generate the random variables Y_j and then add them to obtain a realization of the random variable X. We can apply this method, for example, to obtain Erlang distributed random variable. Note that an Erlang(k) distributed random variable with rate λ can be generated slightly more efficiently by exploiting the fact that

$$\sum_{i=1}^{k} (-\log(u_i)/\lambda) = -\log\left(\prod_{i=1}^{k} u_i\right)/\lambda.$$

However, when k, the number of phases of the Erlang distribution, is large this may be not the most efficient way to generate an Erlang distributed random variable.

4.6 Generating Normal Random Variables

In this section we will show three methods to generate standard normal random variables. The first method is an approximative method using the central limit theorem. The second and third method are exact methods. The second method uses the acceptance-rejection method. The third method is a method developed by Box and Muller which generates two independent standard normal random variables by a transformation of two independent uniform random variables.

Method 1: Using the Central Limit Theorem

An easy, but not very good, approximative method of generating standard normally distributed random variables makes use of the central limit theorem. Recall that the central limit theorem states that for a sequence $Y_1, Y_2, ...$ of independent, identically distributed random variables with mean μ and variance σ^2,

$$\frac{\sum_{i=1}^{n} Y_i - n\mu}{\sigma\sqrt{n}} \xrightarrow{d} N(0,1),$$

where $\xrightarrow{d}$ means convergence in distribution and $N(0,1)$ is a standard normal random variable. If we assume that $Y_i = U_i$, i.e. Y_i is uniformly distributed on (0,1), then we have $\mu = 1/2$ and $\sigma^2 = 1/12$. Hence we can take e.g. $\sum_{i=1}^{12} U_i - 6$ as an approximative standard normal random variable.

Method 2: Using the Acceptance-Rejection Method

The probability density function of the absolute value of a standard normal random value equals

$$f(x) = \frac{2}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}, \qquad 0 < x < \infty.$$

Now, the function

$$g(x) = \sqrt{2e/\pi}\, e^{-x}$$

majorizes the probability density function f. (Check!) We conclude that we can generate a standard normal random variable in the following way: Generate an exponential random variable Y with parameter 1 and a uniform random variable U on the interval $(0,1)$. Accept Y if $U \leq e^{-(Y-1)^2/2}$, otherwise repeat this procedure. Finally, set $X = Y$ or $X = -Y$, both with probability 1/2.

Method 3: Using the Box-Muller Method

Theorem 4.3. *If U_1 and U_2 are two independent, uniformly $(0,1)$ distributed random variables, then*

$$\begin{aligned} X_1 &= \sqrt{-2\ ln\ U_1}\ cos\,(2\pi U_2) \\ X_2 &= \sqrt{-2\ ln\ U_1}\ sin\,(2\pi U_2) \end{aligned}$$

are two independent, standard normally distributed random variables.

Proof: The joint density function of two independent, standard normal random variables (X_1, X_2) equals

$$f(x_1, x_2) = \frac{1}{2\pi} e^{-\frac{1}{2}(x_1^2 + x_2^2)}, \qquad 0 < x_1, x_2 < \infty.$$

Introducing polar coordinates, i.e. $X_1 = R \cos \Phi$ and $X_2 = R \sin \Phi$ we get for the joint density function of R and Φ

$$g(r, \phi) = \frac{1}{2\pi} r e^{-\frac{1}{2}r^2}, \qquad 0 < \phi < 2\pi, \quad 0 < r < \infty.$$

We conclude that R and Φ are independent. Furthermore, it is easy to check that R has the same density as $\sqrt{-2 \ln U_1}$ and Φ has the same density as $2\pi U_2$, where U_1 and U_2 are uniformly $(0, 1)$ distributed. Hence the theorem follows.

5

Fitting distributions

In many practical applications, real data is available for (some of) the random variables in the stochastic simulation model. For example, customer arrival times, claim sizes, service times, or machine up times and down times. In order to be able to simulate such processes, we must find a way to determine the distributions of these random variables, and estimate the parameters of these distributions. This is one of the basic questions in statistics: given a set of samples (data points) $X_1, X_2, \ldots, X_n$, determine which underlying distribution fits the data well, and determine the parameters of this distribution.

In this chapter, we will briefly describe two methods that can be used to estimate the parameters of a given distribution. It is not meant as a complete exposition of what is called *parametric statistics*. Consult any statistical text book for many more techniques of distribution fitting. The last section discusses how to sample from the empirical distribution function.

5.1 Method of moment estimators

Suppose that we have a data set $X_1, X_2, \ldots, X_n$. Suppose that we know that the X_i are drawn from an exponential distribution with unknown parameter λ. The problem of specifying the complete distribution of the X_i is now reduced to specifying the value of λ. From the law of large numbers, we know that the sample mean,

$$\overline{X}_n = \frac{X_1 + X_2 + \ldots + X_n}{n}, \tag{5.1}$$

converges to the expected value (first moment (!)) of the underlying distribution, in this case $1/\lambda$. It therefore seems natural to choose for the parameter λ the value such that

$$\frac{1}{\lambda} = \overline{X}_n, \tag{5.2}$$

so our estimate of λ in this case would be $\widehat{\lambda} = 1/\overline{X}_n$.

In general, the k-th sample moment of $X_1, X_2, \ldots, X_n$ is defined as

$$M_k = \frac{X_1^k + X_2^k + \ldots + X_n^k}{n}, \qquad (k \in \mathbb{N}), \tag{5.3}$$

hence, M_k is the average of the k-th powers of the sample points. Note that the first sample moment is just the sample mean.

Suppose next that our model implies that the X_i are distributed according to the gamma distribution, with unknown parameters $\alpha > 0$ and $\beta > 0$. The gamma distribution has pdf

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}, \qquad x > 0. \tag{5.4}$$

Its mean is given by α/β. Now fitting the gamma distribution to the first sample moment gives the equation

$$\frac{\alpha}{\beta} = M_1. \tag{5.5}$$

A single equation is not sufficient to solve for both α and β, so we must add another equation to the system. The second moment of the gamma distribution is $\alpha(\alpha+1)/\beta^2$. A second equation sets

$$\frac{\alpha(\alpha+1)}{\beta^2} = M_2. \tag{5.6}$$

Now the system of equations (5.5)-(5.6) has a unique solution, which can be found in the following way. From Equation (5.5), it follows that $\beta = \alpha/M_1$. Substituting this into Equation (5.6) yields

$$\frac{\alpha+1}{\alpha} = 1 + \frac{1}{\alpha} = \frac{M_2}{M_1^2}, \tag{5.7}$$

from which it follows that

$$\alpha = \frac{M_1^2}{M_2 - M_1^2}, \tag{5.8}$$

and then, using Equation (5.5), we find

$$\beta = \frac{\alpha}{M_1} = \frac{M_1}{M_2 - M_1^2}. \tag{5.9}$$

Hence, assuming that the data $X_1, X_2, \ldots, X_n$ is distributed according to the gamma distribution, the method of moment estimators results in the following parameters of the distribution:

$$\widehat{\alpha} = \frac{M_1^2}{M_2 - M_1^2}, \qquad \widehat{\beta} = \frac{M_1}{M_2 - M_1^2}. \tag{5.10}$$

Exercise 5.1. Generate a list of 1000 gamma-distributed random variates with parameters $\alpha = 2$ and $\beta = 0.5$. Use the method of moment estimators to estimate the parameters of the gamma distribution, based on the random sample. Create a histogram of the random numbers and compare it to the density of the actual gamma distribution (with $\alpha = 2$ and $\beta = 0.5$) and the fitted distribution.

Solution: See the source code in Listing 5.1 and Figure 5.1.

Listing 5.1: Implementation of the method of moment estimators for the gamma distribution.

```
1  from scipy import stats
2  import matplotlib.pyplot as plt
3  from numpy import arange, mean, sqrt
4
5  # repeat this example for the Gamma distribution
6  n = 1000
7  exactAlpha = 2
8  exactBeta = 0.5
9
10 gammaDist = stats.gamma(exactAlpha, scale=1/exactBeta)
11 data = gammaDist.rvs(n)
```

```
plt.figure()    # create a new plot window
plt.hist(data, bins=40, rwidth=0.8, density=True)

# estimate the parameters alpha and beta of the gamma distribution
M1 = mean(data)
M2 = mean(data**2)
estBeta = M1/(M2-M1**2)
estAlpha = M1*estBeta
fitGammaDist = stats.gamma(estAlpha, scale=1/estBeta)

# Add theoretical density
xs = arange(min(data), max(data), 0.1)
ys1 = gammaDist.pdf(xs)
ys2 = fitGammaDist.pdf(xs)
plt.plot(xs, ys1, color='red')
plt.plot(xs, ys2, '--', color='red')
plt.legend(['Actual distribution', 'Fitted distribution'])
plt.show()
```

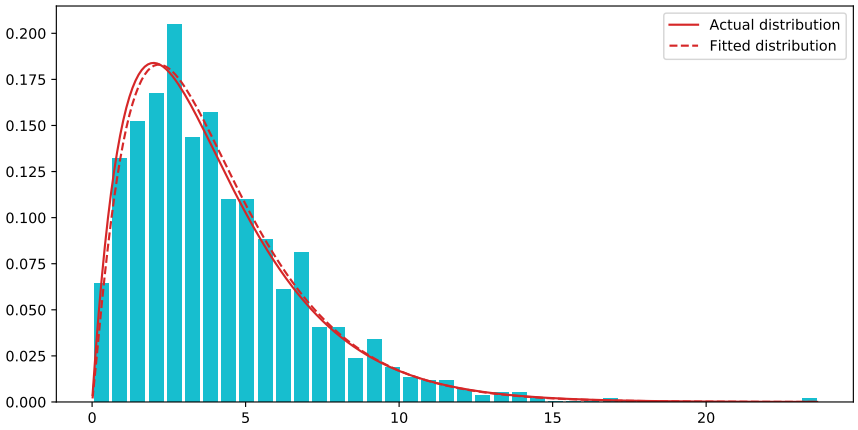


Figure 5.1: Solution to Exercise 5.1.

Exercise 5.2. In a 1905 research[1] into the measurement of Egyptian skulls through the ages, 150 skulls were measured. Part of the data set is displayed in Figure 5.2(a). In the table the maximal breadth of thirty skulls stemming from the year 4000BC are shown. Use the method of moment estimators to fit a normal distribution to this data.

Solution: The first and second moment of the normal distribution are μ and $\sigma^2 + \mu^2$. The method of moment estimators gives rise to the following system of equations:

$$\begin{cases} \mu = M_1, \\ \sigma^2 + \mu^2 = M_2 \end{cases} \tag{5.11}$$

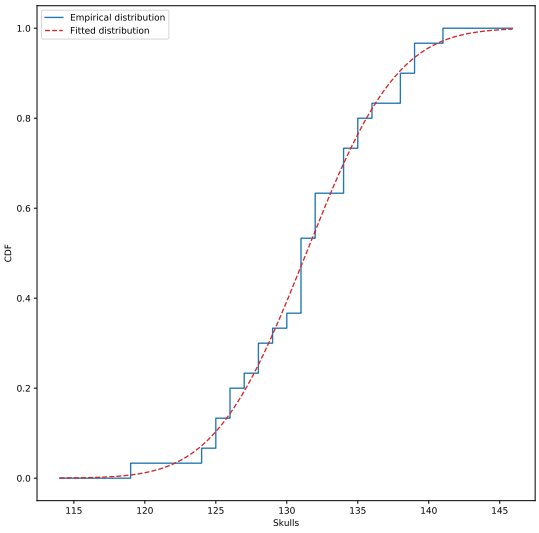
from which it follows that $\widehat{\mu} = M_1$ and $\widehat{\sigma^2} = M_2 - M_1^2$. Substituting the data from Figure 5.2(a), we find that $\widehat{\mu} \approx 131.4$ and $\widehat{\sigma^2} \approx 25.4$. See Listing 5.2.

Listing 5.2: Fitting a normal distribution to the first two moments of the skull data in Figure 5.2(a).

[1] Thomson A. and Randall-Maciver R., *Ancient Races of the Thebaid*, Oxford University Press (1905). Data retrieved from The Data and Story Library.

131	125	131	119
136	138	139	125
131	134	129	134
126	132	141	131
135	132	139	132
126	135	134	128
130	138	128	127
131	124		

(a) Raw data.



(b) Empirical distribution function, compared to the normal distribution plot with parameters estimated by the method of moments.

Figure 5.2: Maximal breadth of 30 Egyptian skulls.

```
print(os.getcwd())  # print the working directory
#os.chdir('..') # change working directory if necessary
import pandas as pd

data = pd.read_csv('skulls.txt', header=0)  # header row is row 0 (the first row)
                                            # Use header=None if there is no header
skulls = data['skulls']  # put the column values in an array

M1 = mean(skulls)       # first moment
M2 = mean(skulls**2)    # second moment
mu = M1
sigma = sqrt(M2 - M1**2)
fitNormDist = stats.norm(mu, sigma)
```

5.2 Maximum likelihood estimation

Another method of parameter estimation is known as maximum likelihood estimation (MLE). Let us assume again that the sample $X_1, X_2, \dots, X_n$ is drawn from the exponential distribution with unknown parameter λ. Their joint distribution is

$$f(X_1, X_2, \dots, X_n|\lambda) = \prod_{i=1}^{n} \left(\lambda e^{-\lambda X_i}\right) = \lambda^n e^{-\lambda \sum_{i=1}^{n} X_i}. \tag{5.12}$$

The function $f(X_1, X_2, \dots, X_n|\lambda)$ is also called the likelihood of the data $X_1, X_2, \dots, X_n$ under the parameter λ. The maximum likelihood estimator for λ is the estimator $\widehat{\lambda}$ that maximises the likelihood of the given data set, hence, we are looking for the value $\lambda > 0$ for which

$$\lambda^n e^{-\lambda \sum_{i=1}^{n} X_i} \tag{5.13}$$

is maximal. Hence, we have to take the derivative (with respect to λ):

$$\frac{\mathrm{d}}{\mathrm{d}\lambda}\lambda^n \mathrm{e}^{-\lambda\sum_{i=1}^n X_i} = \left(n - \lambda\sum_{i=1}^n X_i\right)\lambda^{n-1}\mathrm{e}^{-\lambda\sum_{i=1}^n X_i}. \tag{5.14}$$

From setting the right hand side equal to 0 we find that

$$n - \lambda\sum_{i=1}^n X_i = 0, \tag{5.15}$$

(since λ^{n-1} and $\exp\left(-\lambda\sum_{i=1}^n X_i\right)$ are always positive) which yields the estimator $\widehat{\lambda} = n/\sum_{i=1}^n X_i = 1/\overline{X}_n$.

If a distribution depends on more than one parameter, we have set the partial derivatives with respect to all of these parameters equal to 0. This yields a system of equations, to be solved simultaneously. For example, suppose that we want to fit a normal distribution to the set of data $X_1, X_2, \ldots, X_n$. The density of the normal distribution is

$$f(x|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}}, \tag{5.16}$$

so the likelihood of $X_1, X_2, \ldots, X_n$ under the parameters μ and σ^2 is given by

$$f(X_1, X_2, \ldots, X_n|\mu,\sigma^2) = \left(\frac{1}{2\pi\sigma^2}\right)^{n/2}\exp\left(-\frac{\sum_{i=1}^n (X_i-\mu)^2}{2\sigma^2}\right). \tag{5.17}$$

We would like to take the partial derivatives of this function with respect to both μ and σ^2, but this will be rather difficult. Since the natural logarithm is an increasing function, the likelihood and the so-called *log-likelihood* (the natural logarithm of the joint density function) will attain their maximum values in the exact same point (see Figure 5.3). The log-likelihood is given by

$$\begin{aligned}\Lambda(X_1, X_2, \ldots, X_n|\mu,\sigma^2)) &= \log(f(X_1, X_2, \ldots, X_n|\mu,\sigma^2)) \\ &= \log\left(\left(\frac{1}{2\pi\sigma^2}\right)^{n/2}\exp\left(-\frac{\sum_{i=1}^n (X_i-\mu)^2}{2\sigma^2}\right)\right) \\ &= -\frac{n}{2}\log(2\pi\sigma^2) - \frac{\sum_{i=1}^n (X_i-\mu)^2}{2\sigma^2}.\end{aligned} \tag{5.18}$$

Taking the partial derivatives of the log-likelihood with respect to μ and σ^2 of the log-likelihood gives the following system of equations

$$\begin{cases}\frac{\partial}{\partial\mu}\Lambda(X_1, X_2, \ldots, X_n|\mu,\sigma^2) = \frac{1}{\sigma^2}\left(\sum_{i=1}^n (X_i-\mu)\right) = 0, \\ \frac{\partial}{\partial\sigma^2}\Lambda(X_1, X_2, \ldots, X_n|\mu,\sigma^2) = -\frac{n}{2\sigma^2} + \frac{\sum_{i=1}^n (X_i-\mu)^2}{2\sigma^4} = 0,\end{cases} \tag{5.19}$$

from which the maximum likelihood estimators follow:

$$\widehat{\mu} = \overline{X}_n, \qquad \widehat{\sigma^2} = \frac{1}{n}\sum_{i=1}^n (X_i - \overline{X}_n)^2. \tag{5.20}$$

Note that in this particular case, the maximum likelihood estimators are exactly the same as the estimators we found in Exercise 5.2. In general, this is not the case.

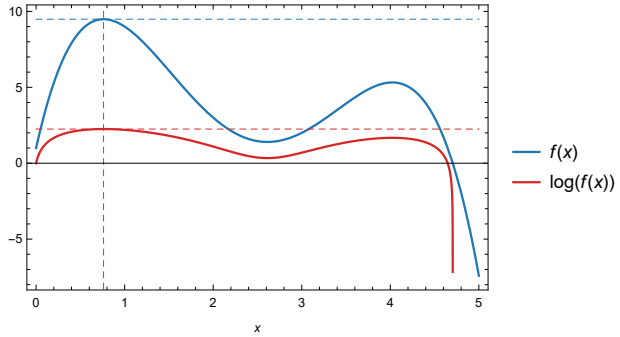


Figure 5.3: A function and its logarithm. The dotted line indicates the maximum of the function, showing that the function and its logarithm assume their maximum on the same value.

Exercise 5.3. Consider again Exercise 5.2. Fit a normal distribution to this data using a maximum likelihood estimator.

Solution: We can use the formulas in Equation (5.20) to determine the estimates for μ and σ^2. In particular, we find $\widehat{\mu} \approx 131.4$ and $\widehat{\sigma^2} \approx 25.4$. Also see the source code in Listing 5.3.

Listing 5.3: Fitting a normal distribution using the maximum likelihood estimators for the mean and variance, based on the data in Figure 5.2(a).

```
muML = mean(skulls)
sigma2ML = mean((skulls - mu)**2)
fitNormDistML = stats.norm(muML, sqrt(sigma2ML))
```

5.3 Empirical distribution functions

Given independent realizations $X_1, X_2, \ldots, X_N$ of some distribution F, we can define a function $\widehat{F}$ as follows

$$\widehat{F}(x) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}_{\{X_i \leq x\}}. \tag{5.21}$$

For each x, the function $\widehat{F}$ gives the fraction of the X_i that does not exceed x. It is therefore a natural approximation to the distribution function F. Indeed, using the strong law of large numbers, it can be shown that $\widehat{F}(x) \to F(x)$ almost surely as $N \to \infty$, for all x. For obvious reasons, the function $\widehat{F}$ is called the *empirical distribution function*.

Python has built-in functionality for obtaining the empirical distribution function, using the command `ECDF` from the `statsmodels.distributions.empirical_distribution` module. An example of how to use this function can be found in Listing 5.4. The script in the listing gives as output a plot such as in the left panel in Figure 5.4, in which the empirical distribution function is compared to the actual distribution function, based on a sample of size $N = 100$ from a standard normal distribution.

Another way of assessing a distribution based on simulations, is the use of histograms or empirical densities. An example can be found in the right panel in Figure 5.4 (also see the last three lines in Listing 5.4).

Sampling from an empirical distribution can be a good alternative to fitting a probability distribution to a given data set. Obviously, this is only a useful technique when sufficient observations are available.

Listing 5.4: Source code for comparison of the empirical distribution function to the actual distribution function of the standard normal distribution.

```
from statsmodels.distributions.empirical_distribution import ECDF

normDist = stats.norm(0, 1)
n = 100
z = normDist.rvs(n)

ecdf = ECDF(z)
plt.figure()
plt.step(ecdf.x, ecdf.y, color='blue', where='post')
xs = arange(min(z), max(z), 0.1)
plt.plot(xs, normDist.cdf(xs), color='red')
plt.legend(['Empirical CDF', 'Theoretical CDF'])
plt.show()
```

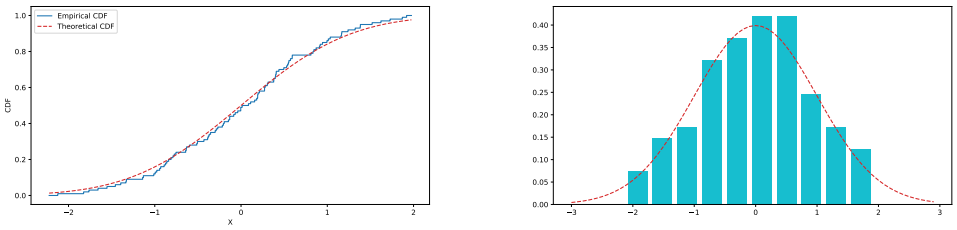


Figure 5.4: Comparison of the empirical distribution function to the CDF and of a histogram to the PDF in a standard normal setting; the results are based on a sample of length 100.

If a data set contains n observations, all we have to do is sample random integers between 1 and n from a discrete uniform distribution and return the corresponding elements from the data set. In Python this can be done efficiently using the `random.choice` function, see Listing 5.5

Listing 5.5: Sampling 10 elements from empirical data.

```
from numpy import random

randomSample = random.choice(skulls, 10, replace=True)
```

5.4 Statistical goodness-of-fit tests

Several statistical tests have been developed for testing whether the fitted distribution and its parameters is adequate. These tests are called *goodness-of-fit tests*, and the most frequently used test is the Kolmogorov-Smirnov test which has already been discussed in Chapter 3. This test is a general test suitable for *any continuous* probability distribution, not just the uniform distribution discussed in Chapter 3. An example of how to apply this test to the sample of skulls is given in Listing 5.6. The P-value of the test is equal to 0.86, which is *much* greater than the commonly used threshold 0.05, indicating that the null hypothesis should not be rejected. The test suggests that the data may very well be normally distributed.

Listing 5.6: The Kolmogorov-Smirnov test and the Shapiro-Wilk in Python, used to check whether the skulls data are normally distributed.

```
tst1 = stats.kstest(skulls, fitNormDist.cdf)
print(tst1)

tst2 = stats.shapiro(skulls)       # Shapiro-Wilk
```

```
5  print(tst2)
```

Another popular test, designed especially to test for normality, is the Shapiro-Wilk test. We do not discuss the details of the test here, but the null hypothesis is that the specified data is normally distributed. Listing 5.6 also includes a Shapiro-Wilk test and the resulting P-value of 0.8603 also suggests that the data could stem from a normal distribution.

5.5 A final disclaimer

As a final remark, we stress that the methods discussed in this chapter are not the state-of-the art. We have decided to discuss these methods because of their popularity in practice, and their simplicity and intuitive nature. In particular, there is much discussion about how distribution fitting should be done correctly. In the literature, for example, it is recommended to use bootstrapping techniques when fitting distributions. Moreover, it is better to choose specific goodness-of-fit tests developed for a specific probability distribution, like the Shapiro-Wilk test for normality, rather than using a general method like Kolmogorov-Smirnov. Statisticians also use Q-Q plots and kernel estimators for the density, rather than histograms and empirical cumulative distribution functions. Still, the methods discussed in this chapter are very frequently used (albeit wrongly, in some cases) and they are sufficient for *our* purpose, which is finding reasonable, approximative distributions to sample input data for our stochastic simulations. It is therefore always recommended to conduct a sensitivity analysis, where the impact of the model parameters (in particular the fitted distributions) on the simulated performance measures is studied.

Part II

Simulation of Stochastic Processes

6

Discrete-time stochastic processes

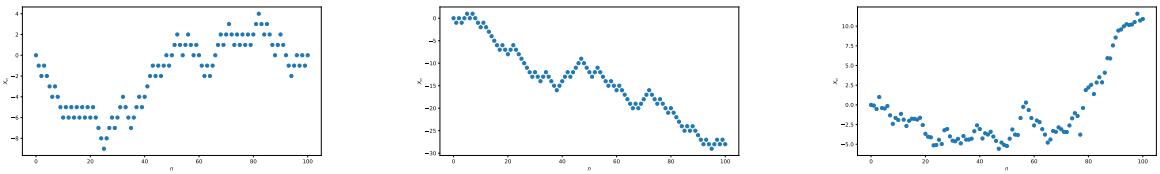
In this chapter we discuss the simulation of some well-known *discrete-time stochastic processes*, which are stochastic processes for which the index variable takes a discrete set of values. In this chapter we discuss random walks and Markov chains. Other well-known examples are branching processes and many models in time series analysis, such as the ARIMA model. Note that discrete-time stochastic processes do not necessarily have discrete state space. We will discuss this topic in more detail in the next section. Since the processes discussed in this chapter take place in discrete time, we can use straightforward Monte Carlo simulation as in Chapter 1.

6.1 Random walks

A random walk is a stochastic process $\{S_n, n = 0, 1, 2, \dots\}$ defined as follows:

$$S_0 = 0, \quad S_n = \sum_{i=1}^{n} X_i, \qquad n = 1, 2, \dots,$$

where $X_1, X_2, \dots$ are independent identically distributed random variables with $\mathbb{E}[|X_i|] < \infty$. The fact that S_n is only defined at discrete time steps $n = 0, 1, 2, \dots$ makes it a discrete-time stochastic process. A *simple* random walk is the special case where $\mathbb{P}(X_i = 1) = p$ and $\mathbb{P}(X_i = 1) = 1 - p$, meaning that it has a discrete state space $0, \pm 1, \pm 2, \dots$. In general, the X_i can have any discrete or continuous distribution function, making a random walk a popular model for general gambling situations, such as stock prices. In Figure 6.1 we show random sample paths of three random walks: two simple random walks with $p = 1/2$ and $p = 1/3$ respectively, and one random walk with normally distributed X_i.



(a) $p = 1/2$ (b) $p = 3/10$ (c) $X_i \sim N(0, 1)$

Figure 6.1: Sample paths of three random walks. Figure (a) is a simple random walk with $p = 1/2$. Figure (b) is a simple random walk with $p = 3/10$. Figure (c) is a random walk where all the X_i are standard normal random variables.

When simulating a random walk, one usually exploits the recursive relation $S_n = S_{n-1} + X_n$, for $n = 1, 2, \dots$. An implementation in Python of a simple random walk with $p = 0.5$ is given in Listing 6.1. An alternative, slightly faster implementation is to generate the vector of $X_1, \dots, X_n$ first and compute

the cumulative sum (see Listing 6.2). A disadvantage of this latter method is that it can only be used when n can be specified on beforehand. With random walks, an important performance measure is often a *hitting time*, i.e., the number of steps it takes to reach a certain level. Since the number of steps n is unknown on beforehand, a simulation as in Listing 6.1 is preferred.

Listing 6.1: Python code to simulate 100 steps of a simple random walk with $p = 1/2$.

```
rng = random.default_rng()    # The random number generator

# Simulate n steps from a simple random walk
def simRandomWalk(p, n):
    s = zeros(n+1)
    for i in range(1, n+1):
        x = rng.choice([-1, 1], p=[1-p, p])
        s[i] = s[i-1] + x
    return s
```

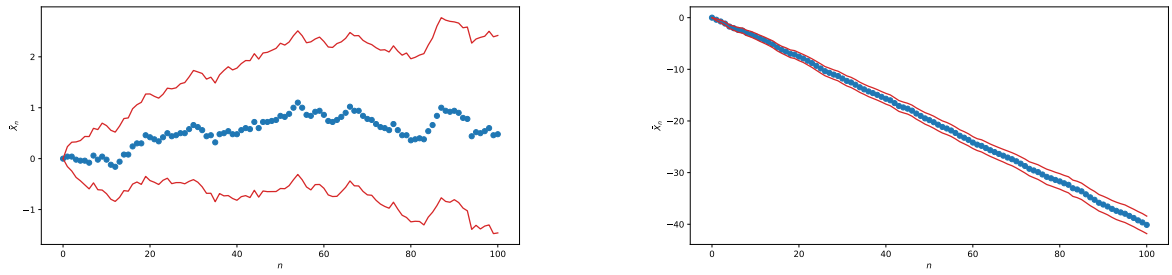
Listing 6.2: Alternative Python code to simulate 100 steps of a simple random walk with $p = 1/2$.

```
def simRandomWalk2(p, n):
    x = rng.choice([-1, 1], p=[1-p, p], size=n)
    s = append([0], cumsum(x))
    return s
```

So far we have focussed on simulating sample paths of random walks. When interested in the expected values $\mathbb{E}[S_n]$, for $n = 1, 2, \ldots$, we should simply take the mean of many independent simulation runs. Of course we can use the Central Limit Theorem to compute approximate 95% confidence intervals, as we have done in chapter 2. An example is shown is Listing 6.3 and the resulting plots are depicted in Figure 6.2. We have created the confidence intervals by keeping track of the sums $\sum_{i=1}^{k} S_i$ and sums of squares $\sum_{i=1}^{k} S_i^2$. Using the relation $\mathrm{Var}[Y] = \mathbb{E}\left[Y^2\right] - (\mathbb{E}[Y])^2$ we have determined the variances needed to compute the half-widths of the confidence intervals. Note that there is a small inaccuracy, due to the fact that the sample variance contains a factor $1/(k-1)$ instead of $1/k$, but for a large number of runs this effect is negligible. Note how the confidence intervals grow wider as n increases. This is due to the fact that

$$\mathrm{Var}[S_n] = \mathrm{Var}\left[\sum_{i=1}^{n} X_i\right] = n\,\mathrm{Var}[X_i],$$

where we have used that the X_i are independent. This means that the width of the confidence intervals grows as $\sqrt{n}$.



(a) $p = 1/2$ (b) $p = 3/10$

Figure 6.2: 95% confidence intervals for a simple random walk, based on 100 replicates.

Listing 6.3: Python code to simulate confidence intervals for simple random walks.

```
nrRuns = 100
n = 100
sumWalks = zeros(n + 1)
sum2Walks = zeros(n + 1)
for i in range(nrRuns) :
    walk = simRandomWalk2(0.3, n)
    sumWalks += walk
    sum2Walks += walk**2

meanWalk = sumWalks / nrRuns              # compute the means
varWalk = sum2Walks/nrRuns - meanWalk**2 # compute the standard deviations
sdWalk = sqrt(varWalk)
ciUpper = meanWalk + 1.96*sdWalk/sqrt(nrRuns)  # Upper CI limit
ciLower = meanWalk - 1.96*sdWalk/sqrt(nrRuns)  # Lower CI limit

# Create a plot
plt.figure()
plt.errorbar(x=arange(0, n+1),
             y=(ciLower + ciUpper) / 2,
             yerr=(ciUpper - ciLower) / 2,
             fmt='o')
plt.show()
```

6.2 Discrete-time Markov chains

A discrete-time Markov chain is a stochastic process $\{X_n, n = 0, 1, 2, \dots\}$ that undergoes transitions from one state to another on a state space. We denote this set of possible states as $\{0, 1, 2, \dots\}$. If $X_n = i$, the process is in state i at time n. The process is memoryless: the transition probabilities to the next state only depend on the current state and not on the events that preceded it:

$$\mathbb{P}(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, X_{n-2} = i_{n-2}, \dots, X_0 = i_0) = P_{ij},$$

for all states $\{i_0, i_1, \dots, i_{n-1}, i, j\}$ and all $n = 0, 1, 2, \dots$. We have $\sum_{j=0}^{\infty} P_{ij} = 1$ for all i. Note that the simple random walk is also a Markov chain with transition probabilities

$$P_{i,j} = \begin{cases} p & \text{if } j = i + 1, \\ 1 - p & \text{if } j = i - 1, \\ 0 & \text{otherwise.} \end{cases}$$

For Markov chains, the following properties are relevant:

- n-step transition probabilities: what is the probability that m steps from now, the process is in state j given that it is currently in state i?
- recurrent and transient states: a state i is *recurrent* if the probability that, starting in state i, the process will ever return to state i is equal to 1. If this probability is less than 1, state i is called *transient*. Note that all states in a simple random walk are only recurrent if $p = 1/2$. Otherwise all states are transient.
- do the limiting probabilities for $n \to \infty$ exist and, if so, what are they?

In the following example we will try to discuss these properties by simulation. Simulating a discrete-time Markov chain can be done by initialising the process at an initial state X_0 and generating the next state by sampling from a discrete random variable as in Section 4.1, where the probabilities $p_1, p_2, \dots$ are the elements in the appropriate row of the transition matrix P. As an example we consider a Markov chain with three possible states $\{1, 2, 3\}$ and transition matrix

$$P = \begin{pmatrix} 0.2 & 0.3 & 0.5 \\ 0.0 & 0.3 & 0.7 \\ 0.5 & 0.4 & 0.1 \end{pmatrix}.$$

A graphical representation of the states and the transitions is depicted in Figure 6.3.

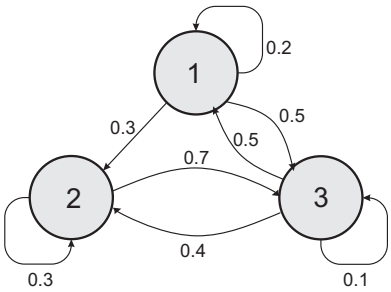


Figure 6.3: The Markov chain discussed in Section 6.2.

Listing 6.4: Python code to simulate 1000 transitions of a Markov chain.

```
rng = random.default_rng()    # The random number generator

# Simulate n transitions of the Markov chain,
# with transition matrix p, starting in state x0
def simMarkovChain(p, x0, n):
    nrStates = len(p)    # = 3
    # Careful here: we need to specify that this array contains integers
    # Using zeros(n+1) would result in an error because x[i-1] is not an integer
    x = zeros(n+1, dtype=int)
    x[0] = x0
    for i in range(1, n+1) :
        x[i] = rng.choice(range(nrStates), p=p[x[i-1]])
    return x

p = array([[0.2, 0.3, 0.5],    # The transition matrix
           [0.0, 0.3, 0.7],
           [0.5, 0.4, 0.1]])
sim = simMarkovChain(p, 0, 1000)
print(sim)
```

A program that simulates this Markov chain (MC) is given in Listing 6.4. We will now use simulation to answer the following questions:

- **If the MC is currently in state 2, what are the probabilities of being in states 1, 2, 3 respectively, four steps from now?**
  In order to answer this question, we have to run four steps of the MC simulation. Simulating multiple runs and counting the number of times that we end in state j, for all j, will give estimates for these probabilities. The simulated values with 10^5 runs are 0.2731, 0.3472, and 0.3797. It is not difficult to determine the theoretical values, since this is simply the values in the second row of the matrix P^3, namely 0.2730, 0.3462, and 0.3808. The R code is given below.

```
nrTransitions = 4
nrRuns = 100000
results = zeros(nrRuns)
for i in arange(nrRuns) :
    sim = simMarkovChain(p, 1, nrTransitions)
    results[i] = sim[-1]  # -1 corresponds to the last element

counts,bins = histogram(results, arange(0, len(p) + 1))
freq = counts/nrRuns
```

- **What are the limiting probabilities of being in each of the states 1, 2, 3 for $n \to \infty$?**
  In order to answer this question, we simply run the simulation for a very long time and count the fraction that the process is in each of the states. Since all of the states are recurrent, it does not matter in which state we start. If we take the simulation length long enough, the

initial state does not impact the estimated fractions. The simulated fractions, with 10^5 steps, are 0.256, 0.340, and 0.405. The theoretical fractions can be determined by computing P^∞, which yields the probabilities 0.254, 0.341, and 0.406. The R code is given below. Note that it would be better not to determine the simulation length on beforehand, but estimate the probabilities during the simulation run and only stop at the moment that the estimated probabilities no longer differ more than a predefined (very small) threshold.

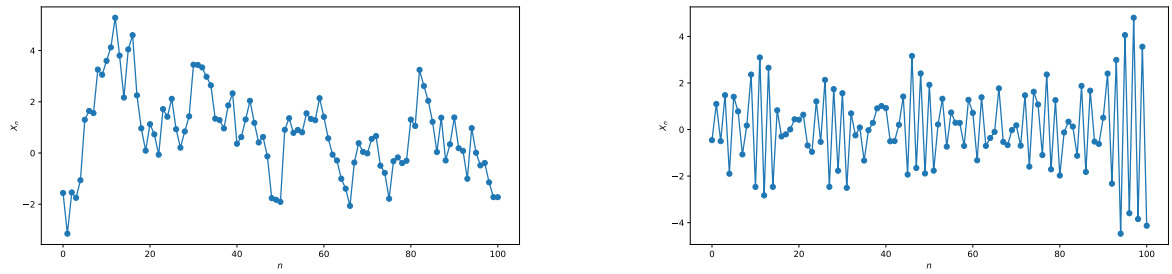
```
nrTransitions = 100000   # just take a very large number of transitions
sim = simMarkovChain(p, 2, nrTransitions)
counts,bins = histogram(sim, arange(0, len(p) + 1))
freq = counts/nrTransitions
```

6.3 The Autoregressive model AR(1)

An first-order autoregressive model, often referred to as an AR(1) model, is a stochastic process $\{X_n, n = 0, 1, 2, \dots\}$ defined as follows:

$$X_0 = Z_0, \quad X_n = \alpha X_{n-1} + Z_n, \qquad n = 1, 2, \dots, \tag{6.1}$$

with $Z_0, Z_1, \dots$ independent random variables with $\mathbb{E}[Z_i] = 0$, $i \geq 0$, and $\text{Var}[Z_i] = \sigma^2$ for $i = 0, 1, 2, \dots$ [1]. The random variables Z_i are often referred to as *white noise*. Note that $\alpha = 1$ results in a random walk with generally distributed steps. Autoregressive models and their extensions are frequently used in time-series analysis. Sample paths for two AR(1) models with $\alpha = 0.8$ and $\alpha = -0.8$ are depicted in Figure 6.4. The positive value of α results in a positively correlated process, whereas the negative value for α results in a (highly oscillating) negatively correlated process. Simulating an AR(1) process



(a) $\alpha = 0.8$ (b) $\alpha = -0.8$

Figure 6.4: Sample paths of two AR(1) processes with standard normally distributed noise. Although this is a discrete-time process, we have connected the dots to illustrate the low and high oscillations.

involves implementing the recursive relation (6.1). See Listing 6.5 for an implementation in Python.

Listing 6.5: Python code to simulate 100 steps of an AR(1) process with $\alpha = 1/2$.

```
def simAR1(alpha, N):
    normDist = stats.norm(0, 1)
    x = normDist.rvs(N + 1)
    for i in range(N):
        x[i+1] = x[i+1] + alpha*x[i]
    return x

n = 100
walk = simAR1(0.8, 100)
```

[1] Some books prefer to define $\text{Var}[Z_0] := \sigma^2/(1-\alpha^2)$, which might have some computational advantages, but we choose the standard definition.

7

Continuous-time stochastic processes

After discussing discrete-time stochastic processes in the previous chapter, we now turn to their continuous-time equivalent. The processes discussed in this chapter vary over time, where time is not modelled as a discrete variable taking integer values, but continuous. Nevertheless, some of these processes can be simulated by a technique quite similar to the discrete simulation discussed in the previous chapter. Other processes require more sophisticated techniques, but they will be discussed in the next chapters.

7.1 The Poisson process

A *Poisson process* is a counting process in which the interarrival times are independent and identically distributed according to an exponential distribution. A stochastic process $\{N(t), t \geq 0\}$ is called a *counting process* if $N(t)$ represents the number of events that have occurred up to time t. This basically means that simulating a counting process involves a counter which is set to zero at time 0 and at each event epoch the counter is increased by one. In case of a Poisson process the inter-event times are independent exponentially distributed random variables. At $t = 0$ the first inter-event time is generated, we increase the current time by this inter-event time, and increase the counter $N(t)$ by one. Then, we wait another exponential amount of time and again increase the counter by one, and so on. A sample path of a Poisson process with rate 0.5 is visualised in Figure 7.1. This simulation technique is called *asynchronous simulation*, which is much more efficient than synchronous simulation for this type of models.

Example 7.1. In this example we simulate a Poisson process with rate 1. Figure 7.1 shows a random, simulated trajectory of such a poisson process. In Listing 7.1 we determine the mean number arrivals in 10 time units. We also estimate the probability that the tenth arrival occurs before time 8.

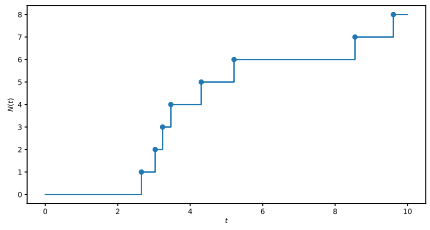


Figure 7.1: A sample path from the Poisson process with rate 1.

For the second part: note that the arrival time of the tenth arrival is the sum of ten independent

exponential random variables with rate 1, so we want to estimate

$$\mathbb{P}\left(X_1 + X_2 + \cdots + X_{10} < 8\right),$$

where $X_i \sim \text{Exp}\,(1)$, independent of all other X_i.

Before we start using simulation to estimate this probability, note that $X_1 + X_2 + \cdots + X_{10}$ follows an Erlang distribution with parameters 10 and 1, as it is the sum of ten exponentially distributed random variables. Thus, the probability can be calculated in an exact way and we find

$$\mathbb{P}\left(X_1 + X_2 + \cdots + X_{10} < 8\right) = 1 - e^{-8} \sum_{i=0}^{9} \frac{8^i}{i!} \approx 0.2834$$

Listing 7.1: Source code used to simulate a Poisson process with rate 1.

```
1  from numpy import zeros, mean, sin, asarray, append, pi
2  from scipy import stats
3  import matplotlib.pyplot as plt
4
5  def simPP(lam, T):    # lambda is a reserved word
6      n = 0
7      expDist = stats.expon(scale=1/lam)
8      t = expDist.rvs()
9      while t < T :
10         n += 1
11         t += expDist.rvs()
12     return n
13
14 nrRuns = 1000
15 arrivals= zeros(nrRuns)
16 tenArrivals = zeros(nrRuns)
17
18 for i in range(nrRuns) :
19     arrivals[i] = simPP(1, 10) # lambda = 1, 10 time units
20     tenArrivals[i] = simPP(1, 8) >= 10
```

To visualise the sample path of (a single simulation run of) a Poisson process, we need to return an array of the actual arrival times. In Listing 7.2 we show how to create such a plot. Since we do not know on beforehand how many arrivals will occur, a data structure that is optimized for variable-sized arrays is preferred. In general, a linked list is a good choice. In Python we use a `deque` (double-ended queue) object. If a function is preferred that generates a fixed number (say n) of arrivals (rather than specifying the simulation time), a more efficient solution is to generate n exponentially distributed random variates and take the cumulative sum to obtain the arrival times.

Simulating a non-homogeneous Poisson process. In practice, we often want to simulate a non-homogeneous Poisson process, i.e., a Poisson process where the rate is time-dependent, $\lambda(t)$. For the simulation of a non-homogeneous Poisson process we can use a thinning approach. Suppose that we want to simulate a non-homogeneous Poisson process with arrival rate $\lambda(t)$ at time t until time T. Assume that $\lambda(t) < \lambda$ for all $t \in [0, T]$. We can simulate this process in the following way: we generate a Poisson process with rate λ and accept a possible arrival at time t with probability $\lambda(t)/\lambda$. This technique is called *thinning* of a Poisson process. In Listing 7.3 we show an example where $\lambda(t) = 2 + 2\sin(\pi t/10)$ and compare it to a regular Poisson process with $\lambda = 2$. Figure 7.2 depicts a sample path of each process.

Simulating a compound Poisson process. A compound Poisson process is similar to an ordinary Poisson process, but at each event, the counter is increased by a *random* variable that may have any

Listing 7.2: Source code used to visualise one sample path of a Poisson process with rate 2 for 100 time units.

```
def simPP2(lam, T):
    arrivalTimes = deque()  # the most efficient data structure for this purpose
    expDist = stats.expon(scale=1/lam)
    t = expDist.rvs()
    while t < T :
        arrivalTimes.append(t)
        t += expDist.rvs()
    return asarray(arrivalTimes)

T = 100
arr = simPP2(2, T)
ys = range(0, len(arr) + 1)
ys = append(ys, [len(arr)]) # point at t=100
xs = append(append([0], arr), [T])  # t = 100
plt.figure()
plt.step(xs,ys, 'b', where='post')
```

Listing 7.3: Simulating a non-homogeneous Poisson process for 100 time units.

```
def lambdat(t):
    return 2 + 2 * sin(0.1*pi*t)

maxLambda = 4
allArrivals = simPP2(maxLambda, 100)
udist = stats.uniform(0, 1)
u = udist.rvs(len(allArrivals))
accept = u * maxLambda < lambdat(allArrivals)
acceptedArrivals = allArrivals[accept]
```

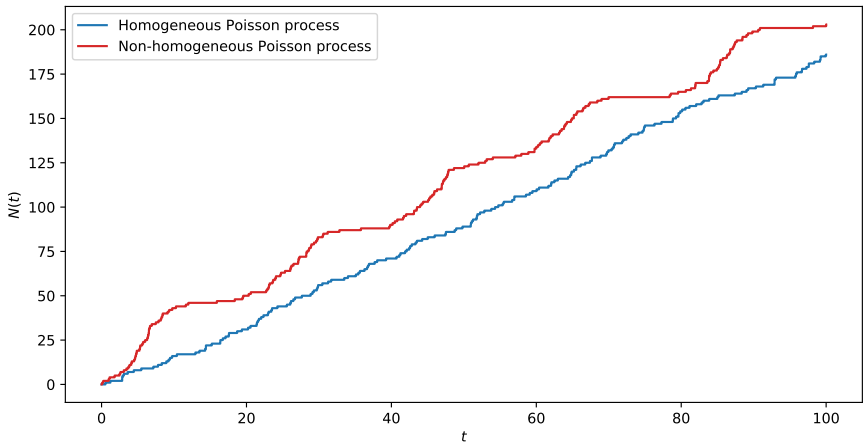


Figure 7.2: Sample paths of a homogeneous and a non-homogeneous Poisson process.

continuous distribution function. These continuous jumps imply that the process takes on real values. A formal definition of a compound Poisson process $\{Y(t), t \geq 0\}$ is:

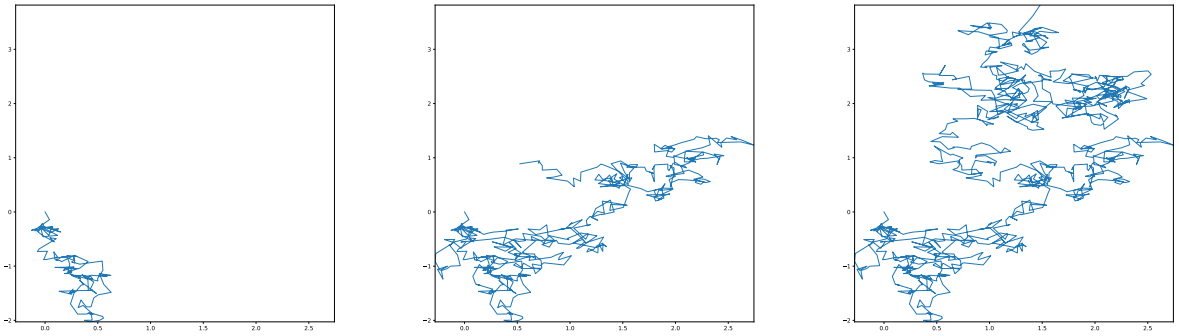
$$Y(t) = \sum_{i=1}^{N(t)} X_i,$$

where $\{N(t), t \geq 0\}$ is a Poisson process and $\{X_i, i = 1, 2, \dots\}$ are iid random variables, independent of $N(t)$. Simulating a compound Poisson process is similar to simulating a Poisson process (see Listing 7.1) but at each event epoch we need to generate a random variable X_i and keep track of the

sum $Y(t)$. More details can be found in Chapter 12 where a practical application is discussed in the area of finance.

7.2 The Brownian motion (Wiener process)

The Brownian motion is one of the most well-known and fundamental stochastic processes. It was first studied in 1827 by botanist Robert Brown, who observed a certain jiggling sort of movement pollen exhibited when brought into water. Later, it was studied by Albert Einstein, who described it as the cumulative effect many small water particles had on the trajectory of the pollen. This is motivated by Figure 7.3, depicting a simulated example of a two-dimensional Brownian motion. The Brownian motion was given a rigorous mathematical treatment by Norbert Wiener, which is why it is often called the Wiener process. For much more information on Brownian motion, see e.g. [Ross96] or the lecture notes by Joe Chang [Chang99]. The Brownian motion is a popular model for stock prices and option pricing. This application is described in more detail in Chapter 14. In this section, we discuss how to simulate the Brownian motion.



(a) Up to $t = 1$ (b) Up to $t = 4$ (c) Up to $t = 8$

Figure 7.3: A simulated path of a 2D Brownian motion, illustrating why Brown used it to model the movement of pollen.

7.2.1 The standard Brownian motion

Definition 7.1. A *standard Brownian motion (SBM)* $\{B(t), t \geq 0\}$ is a continuous-time stochastic process having

(i) continuous sample paths;

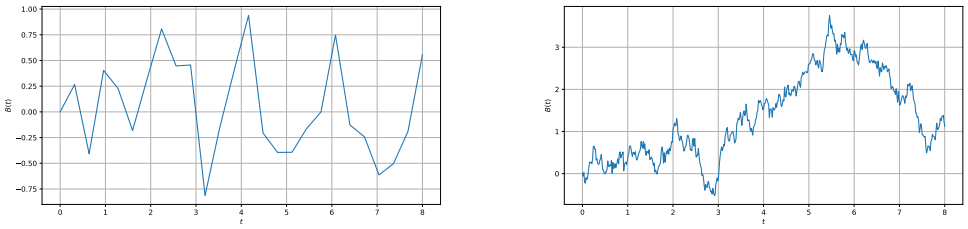
(ii) independent increments (for each $t > s > 0$, $B(t) - B(s)$ is independent of the values $B(u)$ for $0 \leq u \leq s$);

(iii) stationary increments (for each $t > s$, the distribution of $B(t) - B(s)$ depends only on $t - s$);

(iv) $B(0) = 0$;

(v) increments are stationary and normally distributed, that is for each $t \geq s \geq 0$, $B(t) - B(s) \sim \mathcal{N}(0, t - s)$.

Properties *(ii)* and *(iii)* show that if $s < t$, we have $B(t) = Z_1 + Z_2$, where $Z_1 \sim \mathcal{N}(0, s)$ and $Z_2 \sim \mathcal{N}(0, t - s)$. This inspires the following discrete approximation of the Brownian motion.



(a) $M = 25$. (b) $M = 500$.

Figure 7.4: Sample path of the standard Brownian motion.

Let T be our time horizon; we will approximate sample paths from the Brownian motion on the time interval $[0, T]$. Let M be large and let $\tau = T/M$. By definition, for each $i = 1, 2, \dots, M$, we have

$$L_i := B(i\tau) - B\big((i-1)\tau\big) \sim \mathcal{N}(0, \tau), \tag{7.1}$$

and all these L_i are mutually independent. Note that

$$L_1 + L_2 + \dots + L_k = B(k\tau) - B(0) = B(k\tau) \tag{7.2}$$

Thus, we can use $\sum_{i=1}^{k} L_i$ as a discretised approximation to the process up to time $k\tau$. Increasing M makes the grid finer. It should be noted that discrete sampling of the Brownian motion yields just a random walk with $\mathcal{N}(0, \tau)$-distributed increments. We formalise this discussion in the following theorem:

Theorem 7.1. *Let $0 = t_0 < t_1 < t_2 < \dots < t_M = T$ be a subdivision of the interval $[0, T]$, and $Z_1, Z_2, \dots, Z_M \sim \mathcal{N}(0, 1)$ be independent. Define*

$$X_0 = 0, \qquad X_i = X_{i-1} + \sqrt{t_i - t_{i-1}}\, Z_i \quad (i = 1, 2, \dots, M), \tag{7.3}$$

then $(X_0, X_1, X_2, \dots, X_M)$ is equal in distribution to $\big(B_(t_0), B_(t_1), B_(t_2), \dots, B_(t_M)\big)$, where $\{B(t), t \geq 0\}$ is a standard Brownian motion.

A sample path of the standard Brownian motion can be found in Figure 7.4. We will show how to simulate a more general Brownian motion in the next paragraph.

7.2.2 General Brownian motion

Let $\{B(t), t \geq 0\}$ be a standard Brownian motion. A stochastic process $\{S(t), t \geq 0\}$ is called a *Brownian motion* (also (μ, σ^2)-Brownian motion) if it is equal in distribution to

$$S(t) = \mu t + \sigma B(t). \tag{7.4}$$

The parameters μ and $\sigma > 0$ are called the *drift* and the *volatility* of the Brownian motion. For example, if $\mu > 0$, the process has the tendency to increase over time, while a large value of σ increased the variability (volatility) of the process. In Figure 7.5 a sample path of a Brownian motion with $\mu = 2$ and $\sigma = 2$ is depicted.

The simple formula (7.4) shows that if we are able to simulate a standard Brownian motion, then we are also able to simulate a general Brownian motion. Listing 7.4 shows how to simulate a Brownian motion in Python.

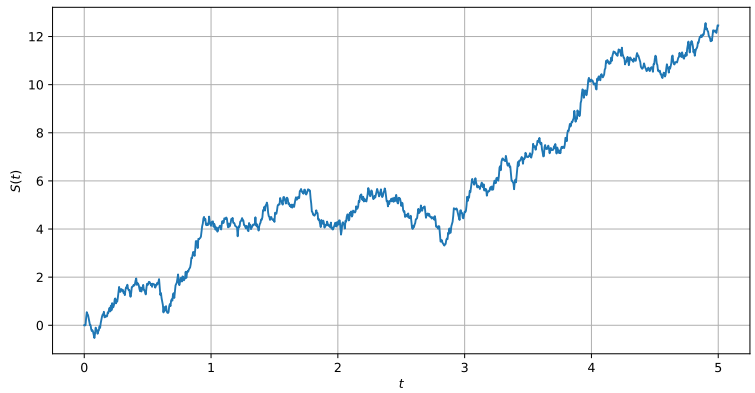


Figure 7.5: Sample path of a Brownian motion with $\mu = 2$ and $\sigma = 2$.

Listing 7.4: Simulation of the general Brownian motion.

```
def simulateBrownianMotion(mu, sigma, T, M):
    dt = T / M
    normDist = stats.norm(mu*dt, sigma * sqrt(dt))
    increments = normDist.rvs(M)
    bm = append([0], cumsum(increments))
    return bm
```

7.2.3 Brownian bridge

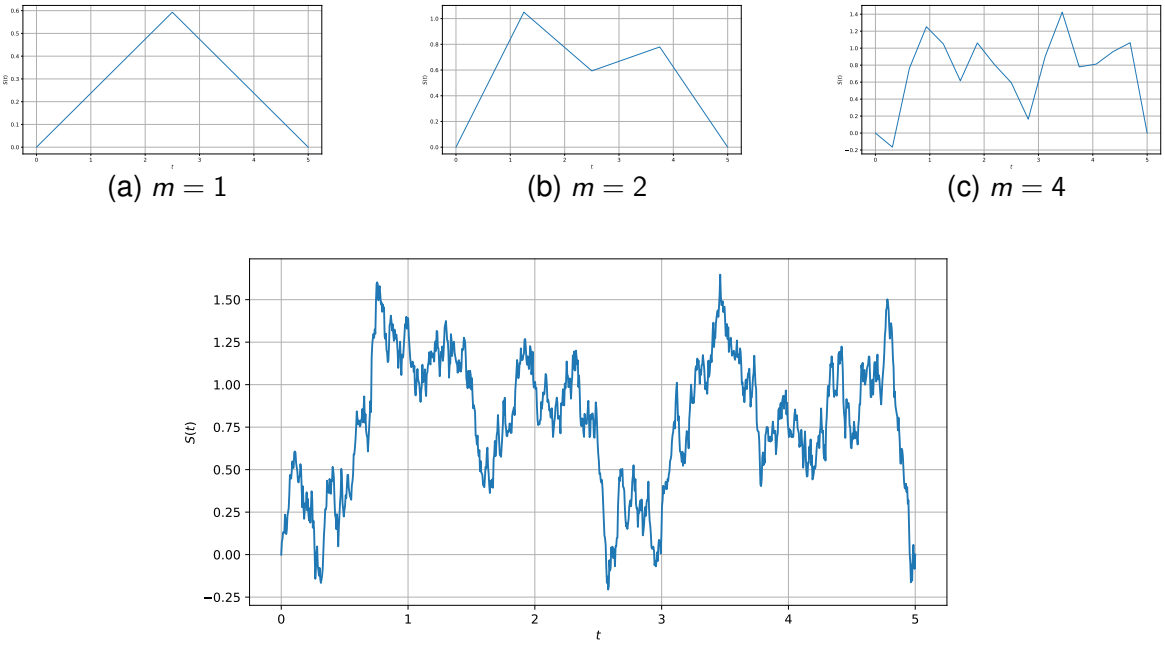
A standard Brownian bridge is a continuous-time stochastic process $\{S(t), 0 \leq t \leq 1\}$ whose probability distribution is the *conditional* distribution of a standard Brownian motion $\{B(t), t \geq 0\}$ *given that* $B(1) = 0$. A general Brownian bridge $\{S(t), 0 \leq t \leq T\}$ can be defined for $0 \leq t \leq T$ using a general Brownian motion $B(t)$ with drift μ and volatility σ given that $B(T) = b$ for an arbitrary value b. It can be shown that this conditional Brownian motion does *not* depend on the drift μ.

A Brownian bridge between $S(0)$ and $S(T)$ (both given) can be simulated by first simulating $S(T/2)$, which can be shown to be normally distributed with mean $\big(S(T) + S(0)\big)/2$ and standard deviation $\frac{1}{2}\sigma\sqrt{T}$. The next step is to simulate a Brownian bridge between $S(0)$ and $S(T/2)$ and another Brownian bridge between $S(T/2)$ and $S(T)$, both conditional on the simulated value of $S(T/2)$. This process can be repeated, first for values $S(T/4)$ and $S(3T/4)$, then $S(T/8)$, $S(3/8)$, $S(5/8)$, and $S(7/4)$, etc. until the desired accuracy is obtained. After m steps, we have simulated a Brownian bridge on a grid of 2^m sub-intervals. The R code to simulate a Brownian bridge is given in Listing 7.5.

Alternative Brownian motion simulation. The scheme to simulate a Brownian bridge can be used to construct a Brownian motion. All one needs to do, is to randomly generate the end point of the Brownian bridge. It can be shown that this value is normally distributed with mean μT and standard deviation $\sigma\sqrt{T}$. Although this implementation certainly has no computational advantages, there are advantages regarding variance reduction techniques which will be discussed later.

7.2.4 Reflected Brownian motion

In many applications a *reflected Brownian motion* is encountered. If $\{B(t), t \geq 0\}$ is a standard Brownian motion, then a Brownian motion reflected at the origin, $\{R(t), t \geq 0\}$, is simply the absolute value of $B(t)$. Simulating a reflected Brownian motion simply involves taking the absolute values of a simulated Brownian motion. However, for a reflected Brownian motion with drift this is not true. In this



(a) $m = 1$ (b) $m = 2$ (c) $m = 4$

(d) $m = 10$

Figure 7.6: Illustration of simulating a Brownian bridge by dividing the interval $[0, T]$ into 2^m subintervals and simulating the values in the centers of the intervals.

situation, it is possible to simulate a Brownian motion with drift and correct every time the BM passes below zero, using a running minimum. An efficient Python implementation is given in Listing 7.6. Unfortunately, it can be shown that this implemenation introduces a small systematic error because due to the discretisation, the minimum is taken over a too small set of values. In these lecture notes we will ignore this effect and refer to Asmussen and Glynn [AG07] for more information and a more accurate simulation scheme.

Listing 7.5: Simulation of a Brownian bridge.

```
def simulateBrownianBridge(b, sigma, T, m):
    M = 2**m
    z = stats.norm(0, 1).rvs(M)
    h = int(M)
    jMax = 1
    w = zeros(M + 1)
    w[M] = b
    sd = sigma*sqrt(T)/2
    for _ in range(m):
        ls = (h*arange(0, jMax)).astype(int)
        h2 = int(h/2)
        lsh2 = (ls + h2).astype(int)
        lsh =  (ls + h).astype(int)
        w[lsh2] = (w[ls] + w[lsh])/2 + sd*z[lsh2]
        sd = sd / sqrt(2)
        jMax *= 2
        h /= 2
    return w
```

Listing 7.6: Simulation of a reflected Brownian motion.

```
def simulateReflectedBrownianMotion(mu, sigma, T, M):
    bm = simulateBrownianMotion(mu, sigma, T, M)
    rbm = bm - minimum.accumulate(bm)  # subtract running minimum
    return rbm
```

7.3 The Ornstein-Uhlenbeck process

The Ornstein-Uhlenbeck process is a continuous-time stochastic process that has a many applications in various fields, such as physics and finance. It can be considered as a continuous-time equivalent of the AR(1) process discussed in Section 6.3. Various representations of this process $\{X(t), t \geq 0\}$ exist, but the one most useful for simulation purposes is a stochastic differential equation:

$$\mathrm{d}X(t) = \theta\big(\mu - X(t)\big)\mathrm{d}t + \sigma \mathrm{d}B(t), \tag{7.5}$$

where $B(t)$ is a standard Brownian motion. A typical feature of the Ornstein-Uhlenbeck process is the drift, which is positive (negative) if $X(t) < \mu$ ($X(t) > \mu$). This means that the drift becomes stronger as the $X(t)$ is further away from its mean, resulting in a tendency to move back towards a central location, with a greater attraction when the process is further away from the centre. This can be seen in Figure 7.7 where sample paths are shown for the same Ornstein-Uhlenbeck process with varying starting values $X(0)$. Since the Brownian motion has independent normally distributed

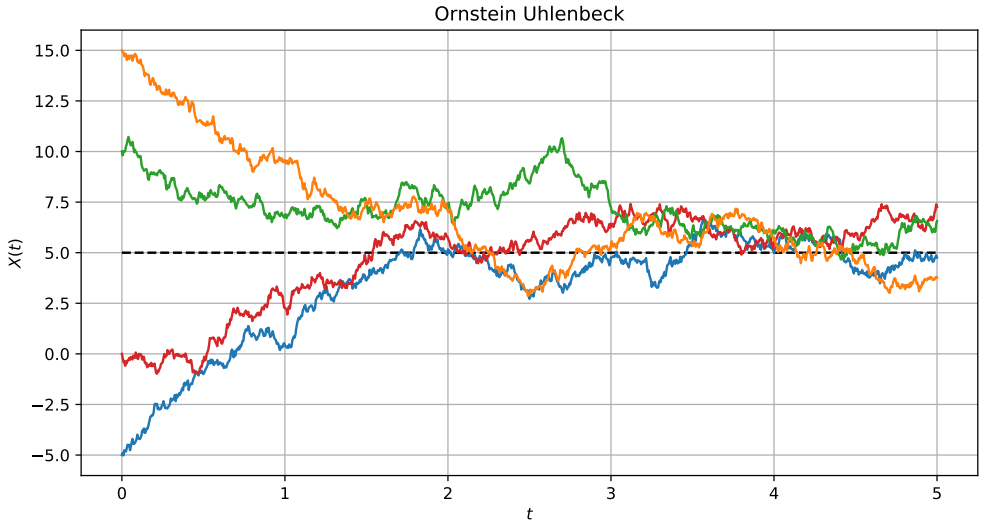


Figure 7.7: Sample paths for the Ornstein-Uhlenbeck process with $\theta = 0.9$, $\mu = 5$, $\sigma = 2$, and different starting values $X(0)$. It can clearly be seen that the process drifts towards its mean, indicated with a dashed black line.

increments, simulating the Ornstein-Uhlenbeck process using discretisation (as we have done for the Brownian motion in Section 7.2) is straightforward, only requiring standard normally distributed random variates. The Python code can be directly deduced from the stochastic differential equation (7.5) and can be found in Listing 7.7.

Listing 7.7: Simulation of the Ornstein-Uhlenbeck process.

```
def simulateOrnsteinUhlenbeck(x0, theta, mu, sigma, T, M):
    dt = T / M    # Grid width
    normDist = stats.norm(0, sqrt(dt))
    dw = normDist.rvs(M)
    x = zeros(M + 1)
    x[0] = x0
    for i in range(1, M + 1) :
        x[i] = x[i-1] + theta*(mu-x[i-1])*dt + sigma*dw[i-1]
    return x
```

7.4 An on-off fluid model

The model discussed in this section is slightly more complicated and is used to illustrate a first example of a discrete-event simulation. Consider a production line consisting of two machines with a finite storage buffer between the two machines (see Figure 7.8).

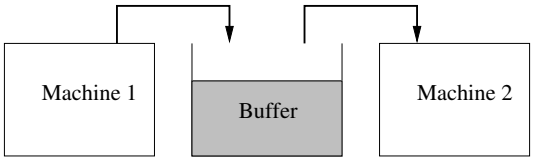


Figure 7.8: Two-machine production line

Machine 1 produces material and puts it into the buffer and machine 2 takes the material out of that buffer. The material is a fluid flowing in and out of the buffer, i.e. we are dealing with a fluid-flow model. The production rate of machine j is equal to r_j, $j = 1, 2$ and we assume $r_1 > r_2$. The first machine is subject to breakdowns. The successive lifetimes and repair times, $X_1, Y_1, X_2, Y_2, \ldots$, are independent random variables with means respectively $\mathbb{E}(X)$ and $\mathbb{E}(Y)$. The second machine is a perfect machine and never breaks down. The size of the storage buffer is equal to K. Clearly, whenever the buffer is full, the production rate of machine 1 is also reduced to r_2. Performance measure of interest is the long-term average production rate of the production line as a function of K. A possible time path realisation for the buffer content is shown in Figure 7.9.

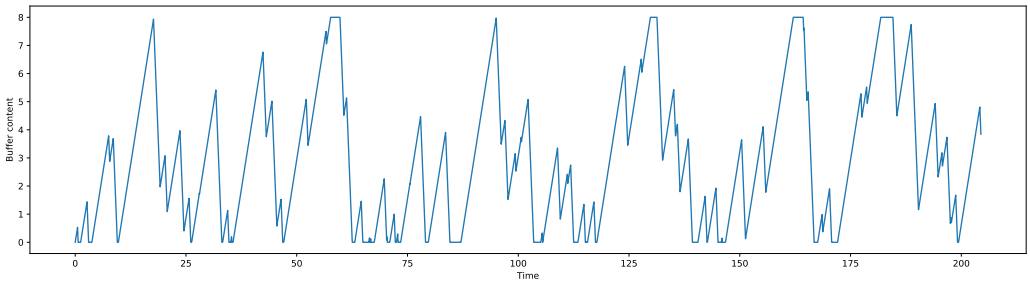


Figure 7.9: Time path realisation of the buffer content

Remark that the behaviour of the system only significantly changes at the moments that machine 1 breaks down and the moments that machine 1 is repaired. At these instants the buffer content switches from increase to decrease or from decrease to increase. In the simulation we jump from one of these so-called event times to the next and calculate the buffer content at these event times, which explains the name *discrete-event simulation*. Once, we know the buffer content at these moments we are able to estimate the average production rate of the production line. If we do a simulation, we obtain one value for the average production rate for one choice of the input parameters. The Python code to simulate this fluid model is given in Listing 7.8.

Listing 7.8: Python code to simulate the fluid model with two machines and one buffer.

```
def simBuffer(lam, mu, r1, r2, K, runLength):
    t = 0           # current time
    b = 0           # current buffer content
    empty = 0       # total time that the buffer is empty
    upDist = stats.expon(scale=1/lam)   # note that this is the MEAN!!!!
    downDist = stats.expon(scale=1/mu)   # note that this is the MEAN!!!!
    while t < runLength :
        u = upDist.rvs()     # up time
        t += u               # update current time
        b = min(b + u * (r1 - r2), K)   # update buffer level after up time
        d = downDist.rvs()   # down time
        t += d               # update current time
        b -= d * r2          # update buffer level after down time
        if b < 0 :
            empty -= b / r2
            b = 0
    return r2 * (1 - empty/t)
```

A big advantage of a simulation is that only by a slight modification of the program we can obtain results for the case in which e.g. the lifetimes or repair times are *not* exponentially distributed. Remark that this is not the case for the analysis because, as mentioned before, we then lose the fact that the process is a continuous-time Markov process. For example, if we change the assumption of exponential repair times to deterministic repair times (with the same mean) a slight modification of our program leads to the outcome 3.923 in the case of buffer size $K = 4$. Hence, simulation is very suitable to investigate whether or not the average production rate is sensitive to the distribution of the repair times and/or lifetimes.

7.5 Continuous-time Markov chains

In this section we discuss simulation of the continuous-time counterpart of the Markov chains described in Section 6.2. We say that the process $\{X(t) : t \geq 0\}$ is a continuous-time Markov chain, if for all $s, t \geq 0$ and nonnegative integers $i, j, x(u), 0 \leq u < s$,

$$\mathbb{P}(X(s+t) = j \,|\, X(s) = i, X(u) = x(u), 0 \leq u < s) = \mathbb{P}(X(s+t) = j \,|\, X(s) = i)\,.$$

We restrict ourselves to time-homogeneous Markov chains, where $\mathbb{P}(X(s+t) = j \,|\, X(s) = i) =: P_{ij}(t)$ is independent of s. These probabilities are the *transition probabilities* of this Markov chain.

To simulate a continuous-time Markov chain, it is more convenient to regard it as a process for which the following holds.

1. Each time it enters state i, the amount of time it spends in that state before making a transition into a different state is exponentially distributed. We denote the rate of this exponential distribution, which only depends on the state i, with λ_i.
2. When the process leaves state i, it next enters state j with probability, say, P_{ij}. Note that $P_{ii} = 0$ and $\sum_j P_{ij} = 1$.

For this reason, a continuous-time Markov chain (CTMC) is usually depicted by its *generator matrix* Q, which contains the transition rates from state i to state j on the non-diagonal elements q_{ij}. The diagonal elements q_{ii} follow from the restriction that the elements of each row should sum to zero (as opposed to 1 for discrete-time Markov chains). The aforementioned parameters can then be easily computed from these matrix elements:

$$\lambda_i = -q_{ii}, \quad P_{ij} = \begin{cases} 0 & i = j, \\ -q_{ij}/q_{ii} & i \neq j. \end{cases}$$

To simulate a CTMC efficiently, we write a discrete-event simulation. The events correspond to the jumps from one state to another. Whenever such an event occurs, we first sample the time until the next event, and subsequently sample the next state. A generic continuous-time Markov chain simulation in Python is given in Listing 7.9. The input parameters are the generator matrix Q, the initial state x_0, and the simulation run length T. The simulation outputs the fraction of time that the simulation was in each state. For long simulation runs, these numbers approximate the steady-state probabilities of the CTMC.

Listing 7.9: Python code to simulate a CTMC with a finite number of states.

```
1  rng = random.default_rng()    # The random number generator
2
3  def simMarkovChain(Q, x0, T):
4      nrStates = len(Q)
5      stateProbs = zeros(nrStates)
6      P = Q.copy()                # create transition probability matrix
7      for i in range(nrStates):   # by copying Q and setting diagonal
8          P[i, i] = 0             # elements to zero...
9      P = P / sum(P, axis=1, keepdims=True)  # ... and dividing by the row sums.
10     t = 0                       # current time
11     state = x0                  # current state
12     while t < T:
13         dt = rng.exponential(scale=1/-Q[state, state]) # time till next event
14         t += dt
15         stateProbs[state] += dt
16         state = rng.choice(range(nrStates), p=P[state])
17     stateProbs = stateProbs / t
18     return stateProbs
```

A small remark about the Python code in Listing 7.9: we have used the function `exponential()` from the Numpy random number generator object instead of our regular solution, which is to use the `stats.expon` object. The reason why we have done this, is because we need to sample one number from a different exponential distribution every time and is probably faster to do it like this. See also the discussion on efficiency of sampling random variates in Appendix B. We will do something similar in Listing 7.10, again with the exponential distribution, but also with the uniform distribution.

Example 7.2 (The $M/M/c$ queue)**.**

In this example, we show how to simulate a queueing system, consisting of one queue of customers that are being served by multiple servers. This example serves as a preparation for the next chapter, which extensively discusses how to simulate queueing systems. The model is depicted in Figure 7.10. The blue rectangles represent customers. They arrive (according to a Poisson process with rate λ) and join the queue. Whenever one of the $c = 3$ servers becomes available, the first and longest waiting customer in the queue will be served by that particular server. Service times are exponentially distributed with rate μ. A graphical representation of this Markov chain, displaying the states and the rates of the transitions between them, is given in Figure 7.11. This queueing system, which in the literature is referred to as the $M/M/c$ queue, is a CTMC, with states $0, 1, 2, \ldots$ representing the number of customers in the system, with the following generator matrix, for $i, j = 0, 1, 2, \ldots$.

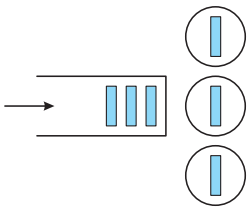


Figure 7.10: The $M/M/c$ queue.

$$q_{ij} = \begin{cases} \lambda & j = i + 1, \\ \min(i, c)\mu & j = i - 1 \text{ and } i > 0, \\ -(\lambda + \min(i, c))\mu & i = j, \\ 0 & \text{otherwise.} \end{cases}$$

Note that this matrix is very sparse, containing many zeros. Moreover, since there is no upper bound

to the number of customers in the system, we have made an adjusted version of Listing 7.9 for this particular model, see Listing 7.10.

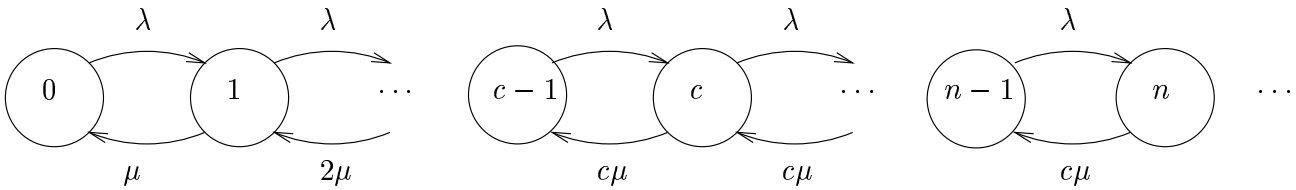


Figure 7.11: The transition diagram, depicting the states and the transition rates of the $M/M/c$ queue.

Listing 7.10: Python code to simulate the queue length distribution of an $M/M/c$ queue.

```
def simMMcQL(lam, mu, c, T):
    maxQL = 100     # keep track of at most 100 queue-length probabilities
    qlProbs = zeros(maxQL + 1)  # empty vector to store these probabilities
    t = 0                       # current time
    n = 0                       # current state
    while t < T:
        lambdai = (lam + min(n, c) * mu)  # lambda_i
        dt = rng.exponential(scale=1/lambdai)  # time till next event
        t += dt
        if n <= maxQL:
            qlProbs[n] += dt
        u = rng.uniform(0, lambdai)    # alternative way to sample
        if u < lam:                 # arrival
            n += 1
        else:                       # departure
            n -= 1
    qlProbs = qlProbs / t
    return qlProbs
```

8

Queueing models

Roughly speaking, there are three different kinds of systems: continuous systems, discrete systems and discrete-event systems. A *continuous system* is a system which state varies continuously in time. Such systems are usually described by a set of differential equations. For example, continuous systems often arise in chemical applications. A *discrete system* is a system which is observed only at some fixed regular time points. Such systems are usually described by a set of difference equations. An example of a discrete system is an inventory model in which we inspect the stock only once a week. The characteristic feature of a *discrete-event system* is that the system is completely determined by a sequence of random event times $t_1, t_2, ...$, and by the changes in the state of the system which take place at these moments. Between these event times the state of the system may also change, however, according to some deterministic pattern.

In this chapter we will demonstrate how to simulate queueing models, which is a general term for mathematical models that involve random arrival and service processes of customers. The main topic of this chapter is the technique called *discrete event simulation*, introduced in Section 8.3. The main example of this chapter, a discrete-event system is a single-server queueing system, is discussed in Section 8.3. A slightly more example involving multiple queues in series is discussed in Section 8.2. In the last section of this chapter we discuss warmup intervals and confidence intervals.

8.1 Discrete simulation of the single-server queue

The basic queueing model is shown in Figure 8.1. It can be used to model, e.g., machines or operators processing orders or communication equipment processing information. Among others, a

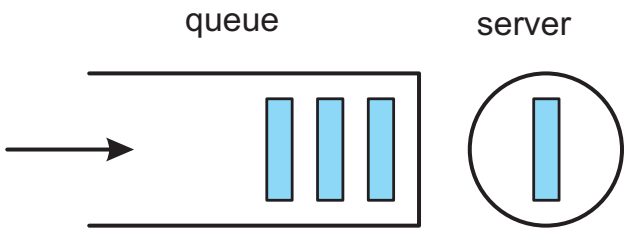


Figure 8.1: A schematic representation of a single-server queue.

queueing model is characterized by:

- The arrival process of customers.
  Usually we assume that the interarrival times are independent and have a common distribution. In many practical situations customers arrive according to a Poisson stream (i.e. exponential

interarrival times). Customers may arrive one by one, or in batches. An example of batch arrivals is the customs office at the border where travel documents of bus passengers have to be checked.

- The behaviour of customers.
  Customers may be patient and willing to wait (for a long time). Or customers may be impatient and leave after a while. For example, in call centers, customers will hang up when they have to wait too long before an operator is available, and they possibly try again after a while.

- The service times.
  Usually we assume that the service times are independent and identically distributed, and that they are independent of the interarrival times. For example, the service times can be deterministic or exponentially distributed. It can also occur that service times are dependent of the queue length. For example, the processing rates of the machines in a production system can be increased once the number of jobs waiting to be processed becomes too large.

- The service capacity.
  There may be a single server or a group of servers helping the customers.

- The waiting room.
  There can be limitations with respect to the number of customers in the system. For example, in a data communication network, only finitely many cells can be buffered in a switch. The determination of good buffer sizes is an important issue in the design of these networks.

Kendall's notation. Kendall introduced a shorthand notation to characterize a range of these queueing models. It is a three-part code $a/b/c$. The first letter specifies the interarrival time distribution and the second one the service time distribution. For example, for a general distribution the letter G is used, M for the exponential distribution (M stands for Memoryless) and D for deterministic times. The third and last letter specifies the number of servers. Some examples are $M/M/1$, $M/M/c$, $M/G/1$, $G/M/1$ and $M/D/1$. The notation can be extended with an extra letter to cover other queueing models. For example, a system with exponential interarrival and service times, one server and having waiting room only for N customers (including the one in service) is abbreviated by the four letter code $M/M/1/N$.

Lindley's recursion. As a first example we will consider the $G/G/1$ queueing system: a single-server queue with customers being served in first-come-first-served order. Suppose that we use a simulation study to obtain an estimator for the long-term average waiting time of customers in this queueing system. First, we remark that we can simulate the waiting time in a $G/G/1$ queue by using a simple discrete simulation instead of a discrete-event simulation. If we denote by W_n the waiting time of the n-th customer, by B_n the service time of the n-th customer and by A_n the interarrival time between the n-th and the $(n+1)$-st customer, then we can obtain the following difference equation for W_n:

$$W_{n+1} = \max(W_n + B_n - A_n, 0). \tag{8.1}$$

This relation, which is called Lindley's recursion, can be used to write a very efficient simulation to estimate the long-term average waiting time of customers. The source code of this discrete simulation can be found in Listing 8.1.

Note that this simulation can be used to estimate the mean, standard deviation, or even the complete distribution of the waiting times. Figure 8.2 depicts a histogram of the simulated waiting times of an $M/M/1$ queue with $\lambda = 0.7$ and $\mu = 0.9$, conditional on the event that they are greater than zero. It can be shown that this conditional waiting time is exponentially distributed with parameter $\mu(1-\lambda/\mu)$. A disadvantage of this simulation is that the queue lengths cannot be simulated without making major adjustments to the simulation.

Listing 8.1: Python code to simulate the $G/G/1$ queue using Lindley's recursion.

```
w = zeros(n)
a = arrDist.rvs(n - 1)  # interarrival times
b = servDist.rvs(n - 1) # service times
for i in range(n - 1):
    w[i+1] = max(w[i] + b[i] - a[i], 0)
return w
```

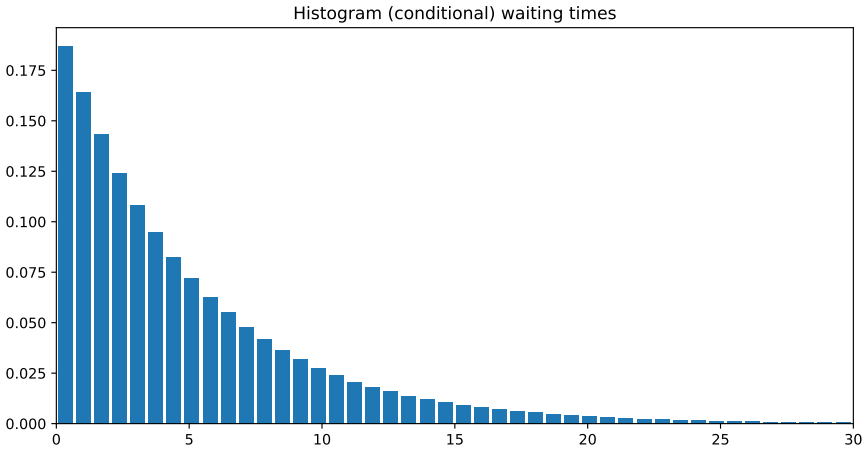


Figure 8.2: A histogram of 10^6 simulated waiting times, conditional on the event that they are greater than 0.

Speeding up the simulation. When implementing the simulations in this chapter, it turns out that programs like Python, R, and Matlab are very slow compared to programming languages like Java and C++. However, this is not always the fault of those programming languages themselves. When writing the simulation code, we often prefer readable code that is easy to understand. This frequently goes at the expense of the simulation speed. Some code might be difficult to speed up, but a general rule of thumb is that avoiding (for) loops tends to improve the speed of the program. As an example, we have provided an alternative implementation of Lindley's recursion in Listing 8.2, replacing the `for` loop by `cumsum` (cumulative sum) and `minimum.accumulate` (running minimum). Personally we find this code less easy to understand, but the reader can verify that it is actually completely equivalent and more than 5 times as fast. In R, a similar implementation speeds up the simulation with a factor greater than 50!

Listing 8.2: More efficient Python code to simulate the $G/G/1$ queue using Lindley's recursion.

```
a = arrDist.rvs(n - 1)  # interarrival times
b = servDist.rvs(n - 1) # service times
d = append([0], b - a)
cumd = cumsum(d)
w = cumd - minimum.accumulate(cumd)  # running minimum
return w
```

Confidence intervals. An interesting question is how to obtain a confidence interval for the mean waiting time. In Chapter 2 we have seen that confidence intervals can be constructed by using multiple independent replications of a single simulation run. However, the simulation results returned by one call to the function `simGG1Lindley` contain *N consecutive* customer waiting times, which are

highly correlated and therefore absolutely *not* suitable to create a confidence interval! In order to obtain *independent* replicates, we should make multiple (say M) calls to the function `simGG1Lindley`. Each call yields one estimate for the expected waiting time, so we can use these M simulated mean waiting times to build a confidence interval. At the end of Listing 8.1 we show how this is done. In Section 8.5 we will discuss this topic in more detail.

8.2 Discrete-event simulation of the multi-server queue: A first approach

So far, we have seen two implementations for the simulation of the single-server queue, but each of them was very specific. Listing 7.10 completely relied on the memoryless property of the exponential distribution and only works for the simulation of *queue lengths* of *Markovian* systems. Listing 8.2 exploited the simple recursive relation (8.1), which makes it only suitable to simulate *waiting times* of a queue with *a singles server*. The purpose of this chapter is to gradually make the simulations more generic, which obviously requires an increase in the complexity. Before turning to the most general simulation for single-server queues, we will write a simulation with the following goals:

- We want to simulate the *queue length distribution*;
- Interarrival times and service times are *generally distributed*;
- We consider a queue with *multiple servers*.

Summarizing these goals: We would like to write an efficient simulation for the queue-length distribution of the $G/G/c$ queue. Although the system is not Markovian, as the $M/M/c$ queue in Example 7.2, the idea behind the simulation will be similar. We will consider the evolution of the queue length. Realising that the queue length only changes at customer *arrival* or *departure* epochs, we identify those two types of events and write a discrete-event simulation that jumps from one event to the next. We use an *object oriented approach* for our program: from now on, the queueing model is regarded as an object with properties (the input parameters) and methods (for example, to simulate one or more runs). Events are also objects, initially with two properties: the type of event (arrival or departure) and the time at which the event takes place. For an improved readability of the code, we also create a separate object to store the simulation results. This object, which will be discussed in more detail below, computes all the relevant performance measures, keeping the code of the main simulation clean. Finally, and this is what separates the examples in this chapter from the earlier discrete-event simulations, we introduce the **future event set**. The future event set (FES) is merely a sorted list of scheduled future events. The reason why we did not need this before, is that we always knew what the next event would be - only the time until the next event was random. Now, depending on the state of the system, the next event can either be an arrival or a departure event. The event scheduling technique using an FES is arguably the most important topic of this chapter, since all efficient simulations of complicated discrete-event systems should employ it. We will now discuss the various objects in more detail.

Event. An event occurs whenever the state of the system (the queue length, in this example) changes. A random sample path of the $G/G/c$ queue is depicted in Figure 8.3. Whenever an arrival occurs, the queue length increases by one, and upon a customer's departure it decreases by one again. In this simple model, we will not look at individual customers (yet), but simply implement a counter keeping track of the current queue length. For this reason, the event class is simple: it only contains a property *type* (arrival or departure) and a property *time* (the time when this event takes place). An implementation in Python of the Event class is shown in Listing 8.3. Note that the Python implementation of Event also contains a function `__lt__`. This is a standard Python function used to compare objects to another ('lt' stands for "less than"), to determine the order in which they should

Listing 8.3: Event class for the $G/G/c$ example in Section 8.2.

```
1  class Event:
2
3      ARRIVAL = 0      # constant for arrival type
4      DEPARTURE = 1    # constant for departure type
5
6      def __init__(self, typ, time):  # type is a reserved word
7          self.type = typ
8          self.time = time
9
10     def __lt__(self, other):        # compare to other events
11         return self.time < other.time
```

be sorted. This method is used by the Future Event Set (see below) to determine how to sort the events. In a discrete-event simulation, they should be ordered by the time at which they occur, which explains the implementation in line 11 of Listing 8.3.

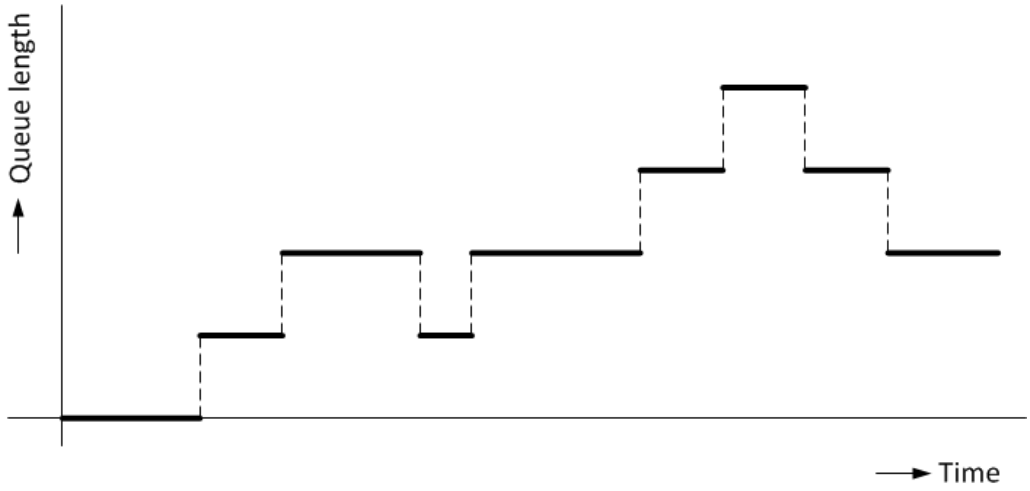


Figure 8.3: A sample path of the queue length of a $G/G/c$ queue.

Future Event Set (FES). This is a variable-sized array where events are added and removed. These events should be sorted by the time of occurrence. Since adding and removing items to a list can be a costly operation, it is important to use a suitable data structure that is optimised for this purpose. In general, this will be a "binary heap" type of structure. Java and Python it is called a priority queue, also referred to as a heap queue. The only methods we need for a future event set, are those required to add and remove events from this list. See Listing 8.4 for a Python implementation. The `heapq` data structure ensures efficient insertion of new events, using the aforementioned `__lt__` function to determine which event comes first.

Simulation results. We encourage creating a separate class to keep track of all the simulation results, mainly because it separates the essential simulation code from the "bookkeeping". It helps the reader to understand the core of the simulation without being bothered by code that is only meant to collect the results. Another advantage is that it becomes easier to write a simulation function that returns multiple performance measures. A third advantage is that (in object oriented programming) you can add functionality to generate various kinds of output to this entity, for example generating histograms and computing confidence intervals.

In this simulation we are interested in the mean queue length and the queue length probabilities. For the latter, we use a list of a predetermined maximum size M (that is chosen so large that the

Listing 8.4: Future Event Set class for the $G/G/c$ example in Section 8.2.

```
import heapq

class FES :

    def __init__(self):
        self.events = []

    def add(self, event):
        heapq.heappush(self.events, event)

    def next(self):
        return heapq.heappop(self.events)
```

probability of having a longer queue length than this specified maximum is negligible). Throughout the simulation, when updating the queue length, we keep track of the amount of time the system has n customers, for $n = 0, 1, 2, \ldots, M$. Denote by $Q(t)$ the queue length (including customer in service) at time t (see Figure 8.3). For the mean queue length, we need to divide the area below the $Q(t)$ by the simulation time. This is formalised in the calculation below:

$$\overline{Q}(T) = \frac{1}{T}\int_0^T Q(t)\mathrm{d}t$$

is the average queue length in $[0, T]$. Define $t_0 := 0$, and let $N(t)$ denote the number of events (arrival or departure) in $(0, t)$. Finally, let $t_1, t_2, t_3, \ldots$ be the event times and let Q_{t_i} define the queue length immediately *after* event i. Then we can write

$$\overline{Q}(T) = \frac{1}{T}\left(\sum_{i=1}^{N(T)} (t_i - t_{i-1})\, Q_{t_{i-1}} + \left(T - t_{N(T)}\right) Q\left(t_{N(t)}\right)\right) \approx \frac{1}{t_{N(T)}}\sum_{i=1}^{N(T)} (t_i - t_{i-1})\, Q_{t_{i-1}},$$

for T large. The code for `SimResults` is shown in Listing 8.5.

Listing 8.5: SimResults class for the $G/G/c$ example in Section 8.2.

```
class SimResults:

    MAX_QL = 10000

    def __init__(self):
        self.sumQL = 0
        self.oldTime = 0
        self.queueLengthHistogram = zeros(self.MAX_QL + 1)

    def registerQueueLength(self, time, ql):
        self.sumQL += ql * (time - self.oldTime)
        self.queueLengthHistogram[min(ql, self.MAX_QL)] += (time - self.oldTime)
        self.oldTime = time

    def getMeanQueueLength(self):
        return self.sumQL / self.oldTime

    def getQueueLengthProbabilities(self) :
        return [x/self.oldTime for x in self.queueLengthHistogram]
```

Main simulation. Having written the classes for event scheduling and computation of the relevant performance measures, we now turn to the main simulation. In a discrete-event simulation, there is always a (usually relatively short) initialisation, where the variables are created, followed by one loop iterating over all the events until some stopping criterion is satisfied. This is usually one of the following: (i) the specified simulation length is exceeded, (ii) the specified maximum number of

customers is reached, or (iii) there are no more scheduled events (for example, because the arrival process stops after some end time).

Initialisation:

- Introduce a variable q to count the number of customers present in the system;
- Introduce a time variable t, initialised to 0;
- Create the SimResults object, to compute the performance measures;
- Create the future event set;
- Generate the first interarrival time, say a, and schedule the first arrival event time a.

In the main loop (in this case until the simulation length exceeds T time units) we determine how and when to schedule all of the events, and how to handle them. It always starts by taking the next event from the FES, set the current time t to the time of this event, and update the performance measures (register the queue length since the previous event). Then we distinguish between the different events.

Arrival event:

- Increase the counter q to include the newly arrived customer;
- If there was an available server upon the customer's arrival, the new customer can be taken into service immediately. In this case: schedule a departure event at time $t + b$, where b is a random service time;
- Generate an interarrival time a and schedule the next arrival event at time $t + a$.

Departure event:

- Decrease the counter q;
- If there was (at least one) customer waiting for service, schedule a departure event at time $t+b$, where b is a random service time.

Tip: An alternative implementation of a renewal arrival process would be to schedule *all* arrival events before handling the first event, instead of scheduling only the first. This would make the remaining code slightly simpler, because one does not have to worry about scheduling arrival events during the core of the simulation. However, it also has some disadvantages. The main disadvantage is that this implementation claims an unnecessarily high amount of computer memory, especially since most simulations may require over one million arrivals. The second disadvantage is that it would be much more difficult to change the arrival process to, for example, a process where the arrival rate depends on the queue length. For this reason, we recommended scheduling only one arrival event at a time.

8.3 Discrete-event simulation of a queueing system: An object oriented approach

Although the previous example is already quite general in its setup, oftentimes an even greater level of detail is required. What is particularly missing so far, are performance measures and different properties for *individual entities* in the system. Typical examples for customers are: priority level, patience, route (in a network), waiting times. For resources (the servers, for example): work speed, occupation level, availability, compatibility with customer types. To implement a simulation for a more complicated queueing system, we recommend using an object oriented approach. In this way, the program is elegantly structured and organised, with each object (entity) storing its own relevant information. Moreover, by creating actual object types for the various entities (customers, servers, other resources) the program becomes more readable and more realistic. The first required step is always to write down the mathematical model, including the notation and all assumptions. In this example, we will skip this step, but we emphasise that this is an extremely important step that cannot be omitted, because the next steps rely on it:

Step 1. Identify the entities (objects) that play a role in the mathematical model and their attributes

Listing 8.6: Main simulation class for the $G/G/c$ example in Section 8.2.

```
from scipy import stats
from dist.Distribution import Distribution
from ch8.ggcQL.Event import Event
from ch8.ggcQL.FES import FES
from ch8.ggcQL.SimResults import SimResults

class GGcSimulationQL :

    def __init__(self, arrDist, servDist, nrServers=1):
        self.arrDist = arrDist
        self.servDist = servDist
        self.nrServers = nrServers

    def simulate(self, T):
        q = 0  # current number of customers
        t = 0  # current time
        res = SimResults()
        fes = FES()
        firstEvent = Event(Event.ARRIVAL, self.arrDist.rvs())
        fes.add(firstEvent)
        while t < T :
            e = fes.next()                      # jump to next event
            t = e.time                          # update the time
            res.registerQueueLength(t, q)       # register the queue length
            if e.type == Event.ARRIVAL :        # if event is an arrival:
                q += 1                          # increase queue length
                if q <= self.nrServers :     # there is an available server
                    dep = Event(Event.DEPARTURE, t + self.servDist.rvs())
                    fes.add(dep)
                arr = Event(Event.ARRIVAL, t + self.arrDist.rvs())
                fes.add(arr)                    # schedule next arrival
            elif e.type == Event.DEPARTURE :    # event is departure
                q -= 1                          # decrease queue length
                if q >= self.nrServers :     # someone is waiting for service
                    dep = Event(Event.DEPARTURE, t + self.servDist.rvs())
                    fes.add(dep)
        return res

arrDist = Distribution(stats.expon(scale=1/2.0))
servDist = Distribution(stats.expon(scale=1/1.0))
sim = GGcSimulationQL(arrDist, servDist, 3)
res = sim.simulate(1000000)
```

(properties) that are relevant to the model. Depending on the implementation and the performance measures of interest, the entities and their attributes may vary. For example, although the servers are typical entities that one might identify in the $G/G/c$ queueing model, it turns out that we do not need to implement it in our simulation program. Typical required entities are *customer*, with attributes like arrival time and service time, and *queue*, with a list of all the waiting customers as attribute.

Step 2. Identify the relevant attributes (properties) for each of these entities can be. In discrete-event simulations, attributes are often times (or the most recent time) at which the state of the entity has changed. For customers, the service time might be a relevant property. Customer type and priority level may also be required properties of the customer entity. The state of an entity may also be a required property for the simulation. For example, whether a machine is up or down, the number of occupied servers, the customers in a queue.

Step 3. Identify the relevant events in the mathematical model. In discrete-event systems, these are the time epochs when the state of the system changes. In the context of queueing models, one should think of (external or internal) arrivals and departures, service beginnings and completions. We will divide this step into two sub-steps. First we determine all events that are relevant for *each entity*. In the second sub-step we determine which of these events take place simultaneously, so-called *compound events*. The final list should not contain multiple events that take place at the same time, as this is irrelevant for the implementation of the simulation model.

Step 4. Identify all performance measures that the program should simulate, and determine which additional properties (for the system or for some of the entities) may be required. We illustrate this in the following example. When the simulation should produce waiting-time estimates, the arrival epoch of each customer should be kept track of. If, however, only queue lengths are relevant, this property is not required at all. Similarly, if the occupation of each server should be simulated, this obviously requires additional properties and possibly even additional entities (such as server, or server pool). This step may seem less important than the previous steps, since oftentimes it can be carried out after most of the simulation has been written. However, this is not always the case and it is therefore a good habit not to skip this step before starting to write the simulation.

Step 5. The last step is to specify the actual implementation of the simulation. This involves describing, in detail, when every event should be scheduled, and how it should be handled. In this stage the role of each entity and its attributes will become apparent. It may occur, however, that certain entities or properties remain unused. In that case the mathematician should decide whether it is better to remove the unused items from the program, or to implement them anyway because they might turn out to be useful in a possible future extension of the program. This should be given careful thought, because extendibility of a program is a very important topic by itself. In the example that we will give in the next paragraph, it turns out that the entity "server" is not required in our implementation. Nevertheless, it may be essential to have a server entity in case we want to simulate a more complicated, extended model, for example including server up- and down-times.

Example 8.1 (The multi-server queue ($G/G/c$).)**.** We will illustrate taking the aforementioned steps in an example where we want to simulate the multi-server queue with general interarrival and service times. In this model customers arrive one by one, with independent identically distributed interarrival times A_i, $i = 0, 1, 2, \ldots$. The service time of customer i, denoted by B_i, is generally distributed. The service times and interarrival times are all independent. There are c identical servers, which serve the customers in order of arrival: first-come-first-served (FCFS). We do not explicitly state how customers are assigned to servers, in case multiple servers are idle, since we are not interested in server-specific performance measures in this example. We stress, however, that in practice this might be a relevant simulation outcome.

Step 1. The two main entities that (seem to) play a role in this model are *customer* and *server*. In order to regulate the queueing process, another entity *queue* is required.

Step 2. For the **customers**, the arrival time is generally a required attribute if the waiting time needs to be determined. The service time and a property indicating whether the customer is waiting in the queue or being served might all be relevant, depending on the implementation. When simulating queueing models with multiple customer classes, obviously the customer class needs to be specified - but in the $G/G/c$ queue this is not the case. When implementing a **server** entity, it is generally a good habit to specify the customer currently being served as a property of the server. This gives more information than merely an attribute indicating whether the server is busy or not. Of course, this customer-being-served attribute may be empty. The main attribute for the entity **queue** is a list of customers currently waiting in the queue, in order of their arrival. In models where service is not FCFS, the order may of course be influenced by, for example, customer priority levels. For regular, FCFS service, a *double-ended queue* (or: deque) data structure is preferred. For priority queues, a *binary heap* is preferred (as we have chosen for the FES).

Step 3. For **customers**, three events may be relevant: the arrival epoch, the moment that a customer is being taken into service (and removed from the queue), and the service completion. For **servers** the relevant events are: service start and service stop. For the **queue** the relevant epochs are whenever customers join or leave the queue. These events are visualised in Figure 8.4. In this figure we observe that we can properly merge some of these events, because they take place simultaneously, and end up with only *two* relevant compound events:

Arrival A customer arrives. This epoch may coincide with the customer joining the queue, or being taken into service if the queue was empty upon his arrival.

Departure A customer's service is completed and he leaves the system. This epoch may coincide with a new customer being removed from the queue and taken into service, but only if the queue was not empty.

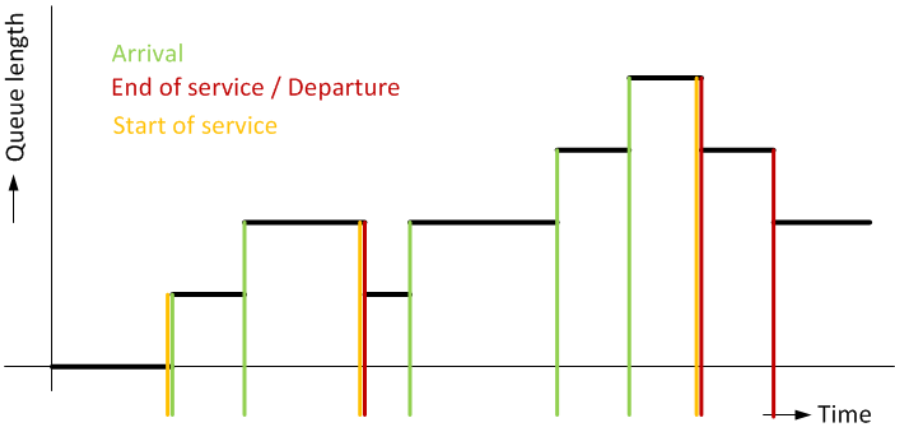


Figure 8.4: Overlapping events in the multi-server queue.

Remark: In our implementation we have chosen to let the Event class have an additional property (besides the type and time of the event): the customer which is associated with the event. This makes it possible to remove the right customer from the queue, upon a departure event. In more complicated models, events might need to have more information about the entities, or resources, or queues that they apply to.

Step 4. In this example we are interested in two performance measures: the empirical queue-length distribution and waiting-time distribution in steady-state. From these, we can deduce other

performance measures like the means and variances of the queue-length and waiting-time distributions, the occupation rate and the fraction that the server is idle. The simplest (but memory consuming) way to determine the empirical waiting-time distribution is to keep track of all simulated customer waiting times and the total time that the system has $0, 1, 2, \ldots$ customers.

Tip: If one is merely interested in the *mean* and *variance* of the waiting-time distribution, it would suffice to keep track of the sum and sum of squares of all waiting times, and the number of customers. Determining the mean and variance of the queue length up to time T can be done by keeping track of the area under the sample-paths of $Q(t)$, the queue length at time t, and $Q^2(t)$ respectively, for $0 \leq t \leq T$ and dividing this area by T. The area only needs to be updated at arrival and departure epochs. Of course it would be simpler to just use Little's law to determine the mean queue length: $\mathbb{E}[Q] = (\mathbb{E}[W] + \mathbb{E}[B])/\mathbb{E}[A]$.

Step 5. Now we can determine how to actually implement the simulation. Already for this simple example, many variations are possible. In order to keep the implementation as straightforward as possible, we make the following design decision: *we will not use the server entity*. This may seem radical at first sight, but we have experienced that most queueing systems do not actually require a server to be implemented as a separate entity. We will simply consider the whole system *including the customer in service* as *one queue*. This only changes the interpretation of the first c customers in the queue, who are not waiting to be served, but actually are *in service*. We have deliberately chosen to make this choice only now, in step 5, because this is how it often goes in practice. When translating the system dynamics into an algorithm or program, it may turn out that efficient implementations do not require all entities or properties to be used. We now describe how and when to schedule all of the events, and how to handle them.

Initialisation:

- Create the FES and the SimResults objects,
- Create the queue entity,
- Introduce a time variable t, initialised to 0,
- Generate the first random interarrival time a and create the first customer c_0,
- Schedule the first arrival event of c_0 at time a.

Code independent of event type:

- Update the time t,
- Let c_1 be customer associated with this event,
- Register the queue length since the previous event.

Arrival event:

- Add c_1 to the queue,
- If the queue length is less than or equal to c, i.e. there was an available server upon arrival:
  - Sample a random service time b for c_1,
  - Schedule a departure event for c_1 at time $t + b$.
  - Register his waiting time (which is zero).
- Generate the next random interarrival time a and create the next customer c_2,
- Schedule the next arrival event of c_2 at time $t + a$.

Departure event:

- Remove c_1 from the queue,
- If the queue length is greater than or equal to c, i.e. there was (at least) one customer waiting for service:
  - Let c_2 be the c-th customer in the queue (in Python: at position $c - 1$),
  - Register the waiting time of c_2, which is $t - c_2$.arrivaltime,

- Sample a random service time b for c_2,
- Schedule a departure event for c_2 at time $t + b$.

An implementation in Python is given in Listings 8.7– 8.10 below.

Event. Note how, compared to Listing 8.3, we now also included a property `customer`, so each event "knows" to which customer it applies. We have also added a function `__str__` which is a standard Python function where one can specify how an object of this class should be printed. This is useful for debugging purposes. Unfortunately, especially in large, complicated simulation models, it can sometimes not be avoided that one needs to debug the code to find an error. Printing each event can be useful in this case.

Listing 8.7: Simulation of the $G/G/c$ queue in Example 8.1: Event

```
class Event:

    ARRIVAL = 0
    DEPARTURE = 1

    def __init__(self, typ, time, cust = None):  # type is a reserved word
        self.type = typ
        self.time = time
        self.customer = cust

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

    def __str__(self):
        s = ('Arrival', 'Departure')
        return s[self.type] + " of customer " + str(self.customer)
        + ' at t = ' + str(self.time)
```

FES. The implementation of the Future Event Set is exactly the same as in Listing 8.4. Probably there is no need to ever change this, for any simulation.

Customer. Compared to the previous example, we now need a Customer entity to keep track of their arrival time, so we can compute their waiting time upon departure. See Listing 8.8.

Listing 8.8: Simulation of the $G/G/c$ queue in Example 8.1: Customer

```
class Customer :

    def __init__(self, arr):
        self.arrivalTime = arr
```

Simulation Results. Compared to the previous example, we now have included functions to register (and return) waiting times. We keep track of all the individual waiting times (in a deque called `waitingTimes`), but additionally we keep track of the sum and the sum of squares of the waiting times. This makes it possible to efficiently compute the mean and the variance. Admittedly, this does not make much sense, given that it is easier to just apply `mean` and `var` to the list of waiting times, but we wanted to illustrate it nevertheless, for cases where the individual waiting times are not recorded. In a similar fashion we also keep track of the squared queue length, to ultimately obtain the variance of the queue length. See Listing 8.9 for a Python implementation.

Listing 8.9: Simulation of the $G/G/c$ queue in Example 8.1: SimResults

```
from collections import deque

from numpy.ma.core import zeros

import matplotlib.pyplot as plt


class SimResults:

    MAX_QL = 10000  # maximum queue length that will be recorded

    def __init__(self):
        self.sumQL = 0
        self.sumQL2 = 0
        self.oldTime = 0
        self.queueLengthHistogram = zeros(self.MAX_QL + 1)
        self.sumW = 0
        self.sumW2 = 0
        self.nW = 0
        self.waitingTimes = deque()

    def registerQueueLength(self, time, ql):
        self.sumQL += ql * (time - self.oldTime)
        self.sumQL2 += ql * ql * (time - self.oldTime)
        self.queueLengthHistogram[min(ql, self.MAX_QL)] += (time - self.oldTime)
        self.oldTime = time

    def registerWaitingTime(self, w):
        self.waitingTimes.append(w)
        self.nW += 1
        self.sumW += w
        self.sumW2 += w * w

    def getMeanQueueLength(self):
        return self.sumQL / self.oldTime

    def getVarianceQueueLength(self):
        return self.sumQL2 / self.oldTime - self.getMeanQueueLength()**2

    def getMeanWaitingTime(self):
        return self.sumW / self.nW

    def getVarianceWaitingTime(self):
        return self.sumW2 / self.nW - self.getMeanWaitingTime()**2

    def getQueueLengthHistogram(self) :
        return [x/self.oldTime for x in self.queueLengthHistogram]

    def getWaitingTimes(self):
        return self.waitingTimes

    def __str__(self):
        s = 'Mean queue length: '+str(self.getMeanQueueLength()) + '\n'
        s += 'Variance queue length: '+str(self.getVarianceQueueLength()) + '\n'
        s += 'Mean waiting time: '+str(self.getMeanWaitingTime()) + '\n'
        s += 'Variance waiting time: '+str(self.getVarianceWaitingTime()) + '\n'
        return s

    def histQueueLength(self, maxq=50):
        ql = self.getQueueLengthHistogram()
        maxx = maxq + 1
        plt.figure()
        plt.bar(range(0, maxx), ql[0:maxx])
        plt.ylabel('P(Q = k)')
        plt.xlabel('k')
        plt.show()

    def histWaitingTimes(self, nrBins=100):
        plt.figure()
        plt.hist(self.waitingTimes, bins=nrBins, rwidth=0.8, density=True)
        plt.show()
```

Listing 8.10: Simulation of the $G/G/c$ queue in Example 8.1: Main program

```
from collections import deque
from scipy import stats
from dist.Distribution import Distribution
from ch8.ggcsim.Customer import Customer
from ch8.ggcsim.Event import Event
from ch8.ggcsim.FES import FES
from ch8.ggcsim.SimResults import SimResults

class GGcSimulation :

    def __init__(self, arrDist, servDist, nrServers=1): # constructor
        self.arrDist = arrDist
        self.servDist = servDist
        self.nrServers = nrServers

    def simulate(self, T):
        fes = FES()                             # future event set
        res = SimResults()                      # simulation results
        queue = deque()                         # the queue
        t = 0                                   # current time
        c0 = Customer(self.arrDist.rvs())       # first customer
        firstEvent = Event(Event.ARRIVAL, c0.arrivalTime, c0)
        fes.add(firstEvent)                    # schedule first arrival event
        while t < T :                           # main loop
            e = fes.next()                      # jump to next event
            t = e.time                          # update the time
            c1 = e.customer                 # customer associated with this event
            res.registerQueueLength(t, len(queue))  # register queue length
            if e.type == Event.ARRIVAL :    # handle an arrival event
                queue.append(c1)            # add customer to the queue
                if len(queue) <= self.nrServers : # there was a free server
                    res.registerWaitingTime(t - c1.arrivalTime)
                    dep = Event(Event.DEPARTURE, t + self.servDist.rvs(), c1)
                    fes.add(dep)            # schedule his departure
                c2 = Customer(t + self.arrDist.rvs()) # create next arrival
                arr = Event(Event.ARRIVAL, c2.arrivalTime, c2)
                fes.add(arr)   # schedule the next arrival
            elif e.type == Event.DEPARTURE : # handle a departure event
                queue.remove(c1) # remove the customer
                if len(queue) >= self.nrServers : # someone was waiting
                    c2 = queue[self.nrServers-1] # longest waiting customer
                    res.registerWaitingTime(t - c2.arrivalTime)
                    dep = Event(Event.DEPARTURE, t + self.servDist.rvs(), c2)
                    fes.add(dep)  # schedule this departure
        return res

arrDist = Distribution(stats.expon(scale=1/2.4)) # interarrival time distr.
servDist = Distribution(stats.expon(scale=1/1.0)) # service time distribution
sim = GGcSimulation(arrDist, servDist, 3) # the simulation model
res = sim.simulate(100000)  # perform simulation
print(res)  # print the results
res.histQueueLength()  # plot of the queue length
res.histWaitingTimes()  # histogram of waiting times
```

Main simulation. The main simulation, following the steps described earlier, is shown in Listing 8.10.

Example 8.2 (Customer impatience in a network of multiple queues in series)**.** The goal of this example is to simulate a more complicated queueing model, a network of N multi-server queues in tandem. After being served in queue i, a customer is transferred to queue $i+1$ or leaves the network if $i = N$. A complicating factor in this model is that customers are impatient. If the time spent in the system is longer than a certain (random) threshold, the customer immediately abandons the system, even if he is being served. A graphical representation of this network is given in Figure 8.5. We will describe the mathematical model and the relevant parameters first, and continue by following the five steps from the previous section. We will not show the complete Python source code, but only the most relevant parts. The complete code is available for download.

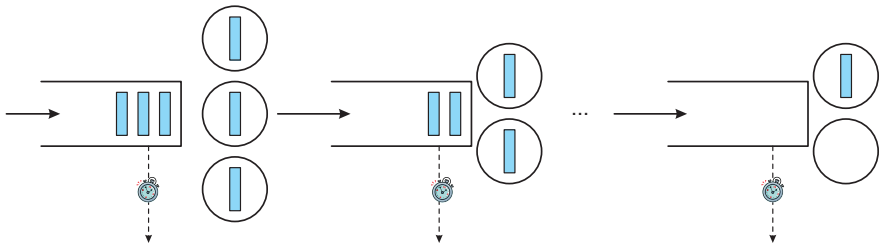


Figure 8.5: A schematic representation of the tandem queues in Example 8.2.

Mathematical model. The system consists of N queues, numbered $1, 2, \ldots, N$. Queue i had n_i servers, $i = 1, 2, \ldots, N$ that serve the customers in FCFS order. Customers arrive at queue 1 according to a Poisson process with rate λ. After a service completion at queue i, customers join queue $i+1$, or leave the system if $i = N$. The transfer of customers from one queue to the next requires no time. The service times of customers at queue i are random variables that are independent of each other, and independent of all other random variables in the model (service times at the other queues and interarrival times). Upon arrival in the system, customers initiate a random timer. If the customer's timer expires before the customer has exited the system, he will abandon the system immediately. If the abandoning customer was in service upon his timer expiration epoch, the first customer waiting to be served will be taken into service (if a waiting customer is present).

The simulation. We will follow the five steps from the beginning of this section to build a simulation model.

Step 1. The model consists of multiple queues, but no new entities have been introduced compared to the single-server queueing model from the previous section. Hence, the entities that we need are **customer** and **queue** to implement the model, and **simulation results** to keep track of all the relevant performance measures.

Step 2. We now describe the relevant attributes for all entities. **Customer:** system arrival time, arrival time at the current queue, service time at the current queue, and patience expiration time. It will also turn out to be useful for customers to know in which queue they are currently residing.

For each **queue** entity we still use a `deque` data type, containing a list of the waiting customers.

Step 3. The relevant events are: arrival in the system (in queue 1), service completion at each of the queues, customer joins the next queue, customer leaves the system, customer impatience runs out. Since we have multiple queues and multiple servers per queue, the **event** entity

Listing 8.11: Network simulation in Example 8.2: Customer

```
class Customer :

    def __init__(self, arrivalTime, serviceTime, endPatienceTime):
        self.arrivalTime = arrivalTime
        self.serviceTime = serviceTime
        self.systemArrivalTime = arrivalTime
        self.endPatienceTime = endPatienceTime
        self.location = 0

    def moveTo(self, location, time, newServiceTime):
        self.location = location
        self.arrivalTime = time
        self.serviceTime = newServiceTime

    def leaveSystem(self):
        self.location = -1
        self.serviceTime = -1
```

needs additional attributes. Note that each event concerns a specific customer, which is the reason that we have chosen to make customer an attribute of the event entity. We can find out where the event takes place, since each customer knows in which queue he is waiting/being served. Again, we have compound events, resulting in a simulation that only needs to distinguish between **arrival**, **departure (at each queue)**, and **abandonment** (impatience runs out) events.

Listing 8.12: Network simulation in Example 8.2: Event

```
class Event:

    ARRIVAL = 0
    DEPARTURE = 1
    ABANDONMENT = 2

    def __init__(self, typ, time, cust):
        self.type = typ
        self.time = time
        self.customer = cust

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

Step 4. The **simulation results** are now stored in a `NetworkSimResults` data type, which contains performance measures on a system level (like sojourn times, total number of abandoned customers), and performance measures per queue. We reuse the `SimResults` class from the previous example, but we add a property to store the number of abandonments for this queue.

Step 5. Before discussing the actual implementation of the simulation, we have to make one more design decision: how to deal with the abandonments? It is clear that upon the entry of a customer in the system, a random patience time should be generated. Consequently, an **abandonment** event is scheduled. It is possible, however, that some of the customers leave the system after being served in each of the queues *before* their patience runs out. This means that the customer is no longer present when the abandonment event is handled. Conversely, it is also possible that **departure** event is scheduled for a customer in service, who will abandon the system during his service. In this case, the corresponding departure event will refer to a customer no longer present in the system. There are (at least) two ways to deal with this issue. The first, possibly most logical way, is to remove the abandonment event for each customer leaving the system, and to remove the departure event for each customer abandoning the system whilst in service. However, we have chosen another option which

Listing 8.13: Network simulation in Example 8.2: (part of) NetworkSimResults

```
class NetworkSimResults:

    def __init__(self, nrOfQueues, nrOfCustomers):
        self.nS = 0
        self.sojournTimes = zeros(nrOfCustomers)
        self.nrOfAbandonments = 0
        self.queueResults = [None] * nrOfQueues
        for i in range(nrOfQueues):
            self.queueResults[i] = SimResults()

    def registerWaitingTime(self, queue, w):
        self.queueResults[queue].registerWaitingTime(w)

    def registerSojournTime(self, s):
        if self.nS < len(self.sojournTimes) :
            self.sojournTimes[self.nS] = s
            self.nS += 1

    def registerAbandonment(self, queue):
        self.nrOfAbandonments += 1
        self.queueResults[queue].registerAbandonment()
```

may be slightly more efficient. Each customer "knows" hist location. So, after leaving the system (either due to impatience, or because he has been served in all queues), his location property will be set to a value indicating that he is no longer present in the system (-1 in our implementation). At the moment that an abandonment or departure event is handled, we check whether the customer is still present in the system and only then handle the event. Otherwise, the event can simply be ignored. This leads to the following simulation.

Initialisation:

- Create the required objects, like FES, the N queues, the simulation results,
- Introduce a time variable t, initialised to 0,
- Generate the first random interarrival time a,
- Generate the first service time b_0 for queue 0 (Python starts counting at zero),
- Generate the first patience time p_0,
- Create the first customer c_0 with arrival time a_0, service time b_0, abandonment time $a_0 + p_0$, and location 0,
- Create and schedule the first arrival event at time a_0 for customer c_0,
- Create and schedule the first abandonment event at time $a_0 + p_0$ for customer c_0.

Listing 8.14: Network simulation in Example 8.2: Main Simulation (Part 1)

```
n = 0
nrQueues = len(self.servDist)
fes = FES()
res = NetworkSimResults(nrQueues, nrOfCustomers)
qs = [deque() for _ in range(nrQueues)]
t = 0                    # current time
a0 = self.arrDist.rvs()# Create arrival of first customer
b0 = self.servDist[0].rvs()
p0 = self.patienceDist.rvs()
c0 = Customer(a0, b0, a0 + p0)
firstEvent = Event(Event.ARRIVAL, a0, c0)
fes.add(firstEvent)
fes.add(Event(Event.ABANDONMENT, a0 + p0, c0))
```

In the main simulation loop we continuously process the first event in the FES, until the maximum number of customers has been reached. We denote the customer associated with the event by c, his location (queue) by q, and his service time by b. The time t will be updated to the time of the event. The event will only be handled if $q \geq 0$ (otherwise the event applies

to a customer that has already left the system).

Listing 8.15: Network simulation in Example 8.2: Main Simulation (Part 2)

```
while n < nrOfCustomers :
    e = fes.next()
    t = e.time
    c = e.customer
    queueNr = c.location
    if queueNr >= 0:
        q = qs[queueNr]
```

Arrival event:

- Add c to queue q,
- If the queue length of queue q is less or equal to n_q, a free server was available and customer c is taken into service immediately. In this case, a departure event at time $t + b$ should be scheduled and his waiting time (zero) should be registered,
- If $q = 0$ (first queue), it was an external arrival and we have to create the next arrival: generate interarrival time a_1, service time b_1 for queue 1, patience p_1; create a customer with arrival time $t + a_1$, service time b_1, abandonment time $t + a_1 + p_1$. Schedule the arrival event at time $t + a_1$ and the abandonment event at $t + a_1 + p_1$.

Listing 8.16: Network simulation in Example 8.2: Main Simulation (Part 3)

```
if e.type == Event.ARRIVAL:
    q.append(c) # add customer to the queue
    if len(q) <= self.nrServers[queueNr] :
        res.registerWaitingTime(queueNr, 0)
        dep = Event(Event.DEPARTURE, t + c.serviceTime, c)
        fes.add(dep) # schedule his departure
    if queueNr == 0: # external arrival
        a1 = t + self.arrDist.rvs()
        b1 = self.servDist[queueNr].rvs()
        p1 = self.patienceDist.rvs()
        c1 = Customer(a1, b1, a1 + p1)
        fes.add(Event(Event.ARRIVAL, a1, c1))
        fes.add(Event(Event.ABANDONMENT, a1 + p1, c1))
```

Departure event:

- Remove customer c from queue q,
- If $q = N-1$ (last queue), register his sojourn time (t minus the customer's system arrival time), notify the customer that he has left the system (i.e., set his location to -1),
- If $q < N-1$, generate a *new* service time b_2 for this customer at queue $q+1$, notify the customer that he moves to the next queue (i.e., increase his location by one) with new service time b_2, schedule a new arrival event for customer c at time t.
  (We schedule an event at the current time, knowing that it will be placed first in the FES and be the next event to be processed.)
- If the queue length of queue q is greater or equal to n_q, the next customer should be taken into service immediately. Let c_3 be this n_q-th customer (i.e., at position $n_q - 1$) in the queue. Schedule a departure event for c_3 at time $t + b_3$, where b_3 is his service time. Register his waiting time (t minus c_3.arrivaltime).

Abandonment event:

- Let i be the position of customer c in his current queue,
- Remove customer c from queue q,
- Notify c that he has left the system (i.e., set his location to -1),
- Register his abandonment,

Listing 8.17: Network simulation in Example 8.2: Main Simulation (Part 4)

```
elif e.type == Event.DEPARTURE :
    q.remove(c)   # remove the customer from the queue
    if queueNr == nrQueues-1 : # if this is the last queue,
        res.registerSojournTime(t - c.systemArrivalTime)
        c.leaveSystem() # that he leaves the system
        n += 1
    else :   # move to next queue and generate new service time
        b2 = servDist[queueNr + 1].rvs()
        c.moveTo(queueNr + 1, t, b2)
        fes.add(Event(Event.ARRIVAL, t, c))
    if len(q) >= self.nrServers[queueNr]:
        c3 = q[self.nrServers[queueNr] - 1]
        res.registerWaitingTime(queueNr, t - c3.arrivalTime)
        b3 = c3.serviceTime
        dep = Event(Event.DEPARTURE, t + b3, c3)
        fes.add(dep)  # schedule this customer's departure
```

- If $i < n_q$ *and* the queue length of queue q is greater or equal to n_q, customer c was in service at the moment that he abandoned the system and there was a customer waiting in the queue that can now be taken into service. Let customer c_2 be the customer at position $n_q - 1$ and schedule a departure event for c_2 at time t plus his service time. Register his waiting time (t minus the c_2.arrivaltime).

Listing 8.18: Network simulation in Example 8.2: Main Simulation (Part 5)

```
else :  # abandonment
    i = q.index(c) # remember the customer's position
    q.remove(c)
    c.leaveSystem()
    res.registerAbandonment(queueNr)
    if i < self.nrServers[queueNr] and len(q) >= self.nrServers[queueNr]:
        c2 = q[self.nrServers[queueNr] - 1]
        fes.add(Event(Event.DEPARTURE, t + c2.serviceTime, c2))
        res.registerWaitingTime(queueNr, t - c2.arrivalTime)
```

Example 8.3 (A matching queueing model)**.** The queueing system in this example is no network, but it does contain multiple queues and multiple servers. The distinguishing feature of this model is that there are different customer types (each with its own queue) and servers that can serve only some of the customer types. This model is typically found in practice in skill-based systems such as call centers. An example could be that the different customer types represent callers speaking different languages and servers are call center operators that have learned to answer questions in one or more specific languages. A schematic representation of the system and the compatibility between customers and servers is given in Figure 8.6. In this figure, we have depicted four customer types, each with their own queue, and five servers. The compatibility between servers and customers types is indicated by the lines between servers and queues. For example, server 1 can server customers of type 1 and 2.

Mathematical model. The system consists of I customer types, each arriving according to their own (independent) arrival process in their dedicated queue, numbered $1, 2, \ldots, I$. Within a queue, customers are served FCFS. There are J servers, numbered $1, 2, \ldots, J$, each of which can serve a subset of the I customer types. The service time required by server j to serve a (compatible) customer of type i has an generic distribution B_{ij} that depends both on i and j. A typical feature for this model is dat it is possible that there are idle servers *and* waiting customers simultaneously, when all idle servers are incompatible with all waiting customers. Whenever a customer arrives to the system, or a server becomes available after a service completion, we attempt to find a suitable match according to the following criteria:

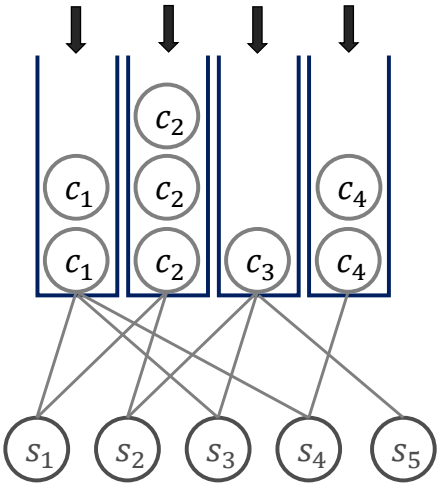


Figure 8.6: A schematic representation of the matching queueing model in Example 8.3.

- If a customer arrives to the system, it searches for a compatible *idle* server. If there is no compatible idle server, it joins the back of its queue. If there is at least one compatible idle server, the server that has been idle longest will serve the newly arrived customer. This is called the Assign to Longest Idle Server (ALIS) policy.

- Whenever a server becomes available after completing a service, it scans *all* queues to find the longest waiting compatible customer. If no such customer can be found, it becomes idle. If there is at least one compatible customer, the longest waiting compatible customer leaves its queue and is served by this server.

The simulation. To implement this model, we first make two extremely important observations: although there are multiple queues with customers, customers are essentially being served according to a global FCFS policy: whenever a server becomes available, it searches for the longest waiting compatible customer. For this reason, it makes sense (and it is much more efficient) to implement *one single queue* for all the customers. This does require, though, that a customer should have a property containing its type.

A second observation is that newly arriving customers search for the longest idle compatible server. The most efficient way to implement this, is to create server entities and a queue which they can join as soon as they become idle and no compatible server can be found in the system. As a consequence, (the implementation of) this system consists of *two queues*: one for the customers and one for the servers (see also Figure 8.7). We will now follow the same five steps from the beginning of this section to build a simulation model.

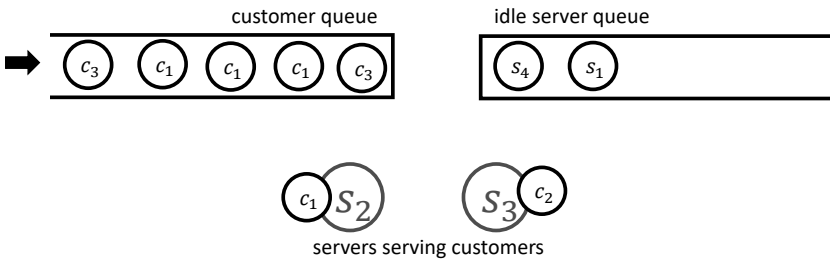


Figure 8.7: An alternative schematic representation of the matching queueing model in Example 8.3.

Step 1. Obviously, we need a **customer** entity that stores its arrival time and type. As argued before, we now also need a **server** entity. There are two **queues**, one for the customers and one for the servers. In both cases, the `deque` data structure in Python is most suitable. Again, a **simulation results** class keeps track of all the relevant performance measures.

Step 2. We now describe the relevant attributes for all entities. **Customer:** arrival time and type. **Server:** stores its number (or id) and knows which customer types it is compatible with. We find it elegant to consider the (probability distribution of the) service time required to serve customers of types $1, 2, \ldots, I$ also as a property of the server entity. Finally, in order to determine the idle times of each server, we need each server to store the start of the current idle time, when idle. This is comparable to customers having to store their arrival times in order to determine their waiting times (idle times are essentially waiting times for a server).

Listing 8.19: Matching simulation in Example 8.3: Customer

```
1  class Customer:
2
3      def __init__(self, arrTime, typ):
4          self.arrivalTime = arrTime     # arrival time
5          self.type = typ                # customer type
```

Listing 8.20: Matching simulation in Example 8.3: Server

```
1  class Server:
2
3      def __init__(self, nr, serviceTimeDists):
4          self.id = nr                                  # server number (or id)
5          self.serviceTimeDists = serviceTimeDists
   # service time distributions
6          self.startIdleTime = 0
   # start of previous idle time
7          nrCustTypes = len(serviceTimeDists)
8          self.compatible = [False] * nrCustTypes    # customer compatibility
9          for i in range(nrCustTypes):
10             if serviceTimeDists[i] != None:
11                 self.compatible[i] = True
```

For each **queue** entity we still use a `deque` data type, containing a list of the waiting customers. One could argue that it is more elegant to write a dedicated class, internally still using a deque, but with added functionality to search for the first compatible element (customer or server). We leave this as an exercise to the reader.

Step 3. The relevant events are: **arrival** and **departure**. Since an arrival corresponds to a specific customer type, and a departure corresponds to a specific customer-server combination, the **event** entity needs the additional attributes *customer* and *server*. Note that arrival events do not contain information about a server.

Step 4. The **simulation results** are stored in a `SimResults` data type, which contains performance measures on a system level (like sojourn times, total number of abandoned customers), and performance measures per queue. Most of this code is similar to what we have seen before, but some of the parameters are slightly different. For example, when registering waiting times, now the customer and the current time is specified. The reason is that this makes it possible to determine the customer's type and store the waiting time in the right location (for that specific customer type). We also keep track of the total idle time per server, which is used to determine the fraction of time that each server was idle.

Step 5. The implementation is described in more detail below.

Initialisation:

Listing 8.21: Network simulation in Example 8.3: Event

```
class Event:

    ARRIVAL = 0
    DEPARTURE = 1

    def __init__(self, typ, time, customer, server=None):
        self.type = typ            # event type
        self.time = time           # event time
        self.customer = customer   # customer associated with this event
        self.server = server       # server associated with this event (optional)

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

Listing 8.22: Matching simulation in Example 8.2: SimResults

```
from numpy import zeros

class SimResults:

    def __init__(self, nrCustTypes, nrServers):
        self.sumQL = 0           # queue length
        self.sumQL2 = 0          # squared queue length (for variance)
        self.tOld = 0            # time of previous event
        self.sumW = zeros(nrCustTypes)  # sum waiting times
        self.sumW2 = zeros(nrCustTypes) # sum squared waiting times
        self.nrCustomers = zeros(nrCustTypes)
        self.sumI = zeros(nrServers)


    def registerWaitingTime(self, t, cust):
        w = t - cust.arrivalTime   # the waiting time
        type = cust.type
        self.sumW[type] += w
        self.sumW2[type] += w*w
        self.nrCustomers[type] += 1

    def registerIdleTime(self, t, server):
        i = t - server.startIdleTime   # the idle time
        nr = server.id
        self.sumI[nr] += i

    def registerQueueLength(self, t, q):
        dt = t - self.tOld
        self.tOld = t
        self.sumQL += dt * q
        self.sumQL2 += dt * q * q

    def getMeanWaitingTimes(self):
        return self.sumW / self.nrCustomers

    def getWaitingTimesVariance(self):
        m2 = self.sumW2 / self.nrCustomers
        m1 = self.getMeanWaitingTimes()
        return m2 - m1*m1


    def getMeanQueueLength(self):
        return self.sumQL / self.tOld

    def getQueueLengthVariance(self):
        m2 = self.sumQL2 / self.tOld
        m1 = self.getMeanQueueLength()
        return m2 - m1*m1

    def getServerIdleTimeFraction(self):
        return self.sumI / self.tOld   # fraction server idle times
```

- Create the required objects, like FES, the queues for the customers and the idle servers (where all servers are stored initially), and the simulation results,
- Introduce a time variable t, initialised to 0,
- Generate the first random interarrival times a_i for customer types $i = 1, 2, \ldots, I$,
- Create the corresponding customers, arriving at these epochs
- Create and schedule the corresponding arrival events.

Listing 8.23: Matching simulation in Example 8.3: Main Simulation (Part 1)

```
1  nrServers = len(self.serviceDist)                  # number of servers
2  nrCustTypes = len(self.arrDist)                    # number of customer types
3  results = SimResults(nrCustTypes, nrServers)       # simulation results
4  fes = FES()                                        # Future Event Set
5  t = 0                                              # current time
6
7  idleServers = deque()                              # queue of idle servers
8  servers = [None] * nrServers                       # list of servers
9  for i in range(nrServers):
10     servers[i] = Server(i, self.serviceDist[i])
11     idleServers.append(servers[i])
12
13 q = deque()                                        # queue of customers
14 for i in range(nrCustTypes):
15     arr = t + self.arrDist[i].rvs()          # create the first arrivals
16     cust = Customer(arr, i)                  # and customers of each type
17     fes.add(Event(Event.ARRIVAL, arr, cust))
```

In the main simulation loop we continuously process the first event in the FES, until the maximum number of customers has been reached. We denote the customer associated with the event by c.

Listing 8.24: Matching simulation in Example 8.3: Main Simulation (Part 2)

```
1  while (t < T):                                     # main loop
2      e = fes.next()                                 # get next event
3      t = e.time                                     # update the current time
4      c = e.customer                                 # customer associated with event
5      results.registerQueueLength(t, len(q))         # register the current total queue
```

Arrival event:

- Search in the idle server queue for the longest idle server compatible with c,
- If this server is not found, add c to the customer queue,
- If there is a compatible idle server, say s, register the waiting time of c (which is zero, by the way) and the idle time of s. Remove s from the list of idle servers, an sample a random service time from B_{ij} (where i is the customer type of c and j is the server id of s). Schedule a departure event.
- Schedule the next arrival event of a type i customer (the same type as c).

Departure event:

- Let s be the server that just completed the service associated with this event,
- Search in the customer queue for the longest waiting customer compatible with s,
- If this customer is not found, add s to the idle server queue and set its "start of idle time" property to t,
- If there is a compatible customer, say *cust*, register the waiting time of *cust* and remove *cust* from the customer queue. Then sample a random service time from B_{ij} (where i is the customer type of *cust* and j is the server id of s). Schedule a departure event.

Example 8.4 (The processor sharing queue)**.** In this example, we discuss a queueing system that is frequently used to model computer systems where a single processor utilises its capacity to serve

Listing 8.25: Matching simulation in Example 8.3: Main Simulation (Part 3)

```
if e.type == Event.ARRIVAL:                    # Handle the arrival event
    s = None
    for i in range(len(idleServers)):          # Find compatible idle server
        s = idleServers[i]
        if s.compatible[c.type]:               # s is compatible with c
            del idleServers[i]                 # remove s from idle servers queue
            break
        else:                                  # s is not compatible with c
            s = None
    if s == None:                              # No compatible server found
        q.append(c)                            # add customer c to the queue
    else:                                      # Compatible server found
        results.registerWaitingTime(t, c)
        results.registerIdleTime(t, s)
        serv = s.serviceTimeDists[c.type].rvs()  # sample random service
        fes.add(Event(Event.DEPARTURE, t + serv, c, s)) # schedule departure
    arr = t + self.arrDist[c.type].rvs()
    c2 = Customer(arr, c.type)
    fes.add(Event(Event.ARRIVAL, arr, c2))     # schedule next arrival event
```

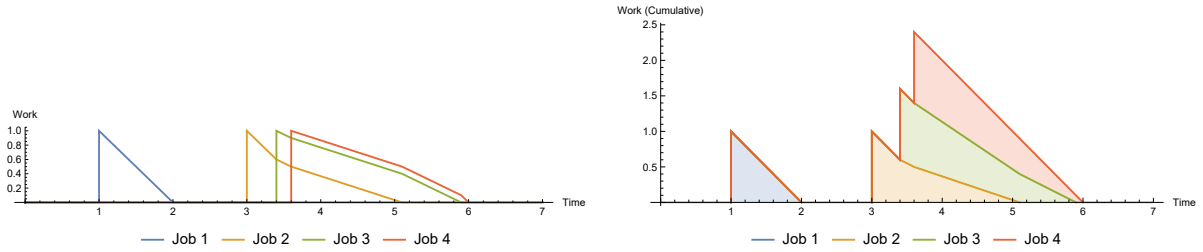
Listing 8.26: Matching simulation in Example 8.3: Main Simulation (Part 4)

```
elif e.type == Event.DEPARTURE:                # Handle the departure event
    s = e.server
    cust = None
    for i in range(len(q)):                    # Find compatible customer
        cust = q[i]
        if s.compatible[cust.type]:            # s is compatible with cust
            del q[i]                           # remove cust from queue
            break
        else:                                  # s is not compatible with cust
            cust = None
    if cust == None:                           # No compatible customer found
        idleServers.append(s)                  # add server s to the idle servers
        s.startIdleTime = t                    # set start of idle time to t
    else:
        results.registerWaitingTime(t, cust)
        serv = s.serviceTimeDists[cust.type].rvs()
        fes.add(Event(Event.DEPARTURE, t + serv, cust, s)) # schedule departure
```

multiple jobs simultaneously, each job receiving an equal share of this capacity: the processor sharing queue. The model is as follows: jobs arrive to the system according to a Poisson process with rate λ and have generic i.i.d. job sizes B_i, $i = 1, 2, \ldots$. Jobs are taken into service immediately, i.e., they have waiting time zero. However, the speed at which a job is served depends on the number of jobs present in the system, simultaneously. For example, when a job of size x enters the system and it remains the only job in the system, it takes x time units to complete this job. If, however, upon arrival, there is one other job in the system, the server works at half speed on both jobs from that moment on. The challenge is to determine the distributions of the queue length (the number of jobs in the system) and the sojourn time, i.e. the time spent in the system by an arbitrary job.

To better understand the system, consider the following example where all jobs have deterministic sizes of $B_i = 1$. Assume that the first four jobs arrive at times $t_1 = 1$, $t_2 = 3$, $t_3 = 3.4$ and $t = 3.6$. Then the completion times can be determined as follows: the first job is the only one in the system, and will be served from time $t = t_1 = 1$ until $t = t_1 + B_1 = 1 + 1 = 2$. The second job enters the system at time $t = 3$ At time $t = 3.4$, 40% of this job has been completed. On that epoch, however, the third job enters the system and both jobs receive 50% of the server's capacity, until the fourth job arrives at time $t = t_4 = 3.6$. At that time, 0.5 units (out of 1.0) of job 2 and 0.1 units of job 3 have been completed. Now, the server serves each of the jobs at 1/3 of its capacity. This means that it takes $0.5 \times 3 = 1.5$ time units to complete job 2. This job leaves the system at time $t = 3.6 + 1.5 = 5.1$. Now the server works at 50% on jobs 3 and 4 (with respectively 0.4 and 0.5 units remaining). Next, job 3

is completed at time $5.1 + 0.4 \times 2 = 5.9$. The remaining 0.1 units of job 4 are completed at time $t = 6$ because it is now the only job in the system. A graphical representation of this example is depicted in Figure 8.8, where the workload is shown versus the time. In particular, it can be seen in Figure 8.8(b) that the *total* workload decreases at unit rate.



(a) Amount of work per job (b) Amount of work per job (cumulative)

Figure 8.8: Visualisation of the amount of work versus time in the processor sharing example.

In this example it became apparent that the distinguishing feature of a processor sharing queue is that the job completion times can change while a job is in the system: every arrival or completion of another job changes the completion times of all jobs in the system. In a discrete event simulation, this implies that event times need to be changed, which is exactly what we generally try to avoid because of efficiency reasons.

Event handling. Again, the only two event types are arrival and departure events. The Poisson (or renewal) arrival process is implemented in the usual way: upon arrival of a job, say at time t, we schedule the next arrival at time $t + a$ where a is a new random interarrival time sample. Since jobs have zero waiting time, because the service starts immediately upon arrival, departure events are scheduled upon arrival too. The departure epoch depends on the job size b (a random sample from the service-time distribution) and the number of jobs on the system. If there are N jobs present in the system at time t, including the newly arrived job, then the service completion is scheduled at time $t + b \times N$, since every job receives a fraction $1/N$ of the server's capacity. Additionally, the completion times of the other $N - 1$ jobs in the system need to be adjusted. Let $c_1^{\text{old}}, \ldots, c_{N-1}^{\text{old}}$ denote the old job completion times (before the arrival of the N-th job). Then the residual service time of job i is $c_i^{\text{old}} - t$, which means that the remaining amount of work for job i is $(c_i^{\text{old}} - t)/(N - 1)$ units. The new completion times will become:

$$c_i = (c_i^{\text{old}} - t) \times N/(N - 1), \qquad i = 1, \ldots, N - 1. \tag{8.2}$$

Following a similar way of reasoning, whenever a job is completed (say at time t, with N jobs in the system), the remaining $N - 1$ job completion times will become:

$$c_i = (c_i^{\text{old}} - t) \times (N - 1)/N, \qquad i = 1, \ldots, N - 1. \tag{8.3}$$

The simulation. Most of the files (simulation results, event, customer) are exactly the same as, or really similar to, the ones we have seen before. The future event set has one additional method, though, which changes the event times of the departure events. Please note the following:

- There is always exactly *one* arrival event in the FES.
- When the departure event times are changed, the *order* of these departure events remains the same. *S*ince the time of the arrival event remains unchanged, the overall order of *all* events, may be different though.

There are numerous ways to implement this, all of them with certain drawbacks and advantages. A straightforward way is to move all events to a new FES, which then ensures correct ordering of the

events. When using an efficient data structure, like the `heapq`, this is not too inefficient. Listing 8.27 shows two possible implementations, which seem to require approximately the same amount of computation time. The parameters `oldQL` and `newQL` denote the old and new queue length, respectively. The first function copies all events to a new FES, which then becomes the actual FES, while the second function changes the departure event times and then sorts them again (heapify). The main

Listing 8.27: Reordering the events in the processor sharing queue: two possible implementations.

```
def updateEventTimes(self, t, oldQL, newQL):
    events2 = []
    for e in self.events:
        if e.type == Event.DEPARTURE:
            e.time = t + (e.time - t) * newQL / oldQL
        heapq.heappush(events2, e)
    self.events = events2

def updateEventTimes2(self, t, oldQL, newQL):
    for e in self.events:
        if e.type == Event.DEPARTURE:
            e.time = t + (e.time - t) * newQL / oldQL
    heapq.heapify(self.events)
```

simulation follows the following steps: **Initialisation:**

- Create the required objects, like FES, the queue for the customers, and the simulation results,
- Introduce a time variable t, initialised to 0,
- Create and schedule the first arrival event.

Arrival event:

- Add customer to the queue,
- Change the departure event times of the jobs that were in the system according to (8.2),
- Schedule the departure of the newly arrived job,
- Create the customer that arrives next and schedule the corresponding arrival event.

Departure event:

- Register the sojourn time of the customer/job that is completed,
- Remove customer from the queue,
- Change the departure event times of the jobs that are in the system according to (8.3).

Remark: Although the main purpose of this example is to illustrate that there might be discrete event systems where scheduled events need to be modified, this remains an inefficient operation and should therefore be avoided whenever this is possible. Naturally, there are more efficient implementations than the one we presented. For example, reminiscent of the efficient queue-length implementation of the G/G/1 queue in Section 8.2, we can use an array of job completion times instead of the FES object with customer objects to speed up the simulation. By treating the arrival event separately from these job completion events, and not trying to keep track of customer waiting times, a significant speedup can be achieved compared to the object oriented approach, in particular when the system load is high. This efficient code is presented in Listing 8.29.

8.4 Transient versus steady-state behaviour

In the previous sections we have been interested in the steady-state waiting-time distribution. In practical situations, it is often more relevant to study the transient (time dependent) behaviour of the system. This is particularly relevant in systems with time-varying arrival rates and systems that only operate for a limited amount of time. Except for production environments that run 24/7, most systems

Listing 8.28: Main simulation of the processor sharing queue.

```
def simulate(self, T):
    fes = FES()                                  # future event set
    res = SimResults()                           # simulation results
    queue = deque()                              # the queue
    t = 0                                        # current time
    c0 = Customer(self.arrDist.rvs())            # first customer
    firstEvent = Event(Event.ARRIVAL, c0.arrivalTime, c0)
    fes.add(firstEvent)                         # schedule first arrival event
    while t < T :                                # main loop
        e = fes.next()                           # jump to next event
        t = e.time                               # update the time
        c1 = e.customer                          # customer associated with this event
        oldQL = len(queue)                       # old queue length
        res.registerQueueLength(t, oldQL)        # register queue length
        if e.type == Event.ARRIVAL :     # handle an arrival event
            queue.append(c1)             # add customer to the queue
            fes.updateEventTimes(t, oldQL, oldQL + 1) # update all event times
            dep = t + self.servDist.rvs() * (oldQL + 1) # determine departure time
            fes.add(Event(Event.DEPARTURE, dep, c1))  # schedule departure
            c2 = Customer(t + self.arrDist.rvs()) # create next arrival
            arr = Event(Event.ARRIVAL, c2.arrivalTime, c2)
            fes.add(arr)    # schedule the next arrival
        elif e.type == Event.DEPARTURE : # handle a departure event
            res.registerSojournTime(t - c1.arrivalTime)
            queue.remove(c1) # remove the customer
            fes.updateEventTimes(t, oldQL, oldQL - 1) # update all event times
    return res
```

involving (human) waiting lines exhibit this kind of behaviour. The main reason to study steady-state performance characteristics, is because analytical, queueing theoretical methods can generally only be applied to steady-state models. Moreover, frequently the time required for a system to reach steady state is short, which makes the steady-state model a suitable choice.

In contrast to an analytical approach, simulating steady-state results is more difficult than simulating the transient behaviour. Firstly, a simulation seldom starts in steady state, which means that there already is a systematic bias when using an average of all simulated values as an estimate for a steady-state performance measure. Second, it would theoretically require an infinitely long simulation run to obtain steady-state results. In practice, this implies that *very long* simulation runs are required. In this section we discuss the transient and steady-state behaviour of the $G/G/1$ queue, introduced in the first two sections of this chapter. We will use exponentially distributed interarrival and service times with parameters $\lambda = 0.7$ and $\mu = 0.9$ respectively.

We define W_n as the waiting time of the n^{th} customer in the system. The steady-state mean waiting time for this $M/M/1$ model can be computed theoretically:

$$\mathbb{E}[W] = \lim_{n\to\infty} \mathbb{E}[W_n] = \frac{\lambda}{\mu(\mu-\lambda)} = 3.8889.$$

We now use simulation to study how fast W_n converges to this limiting value. Figure 8.9 shows a plot of simulated values of $\mathbb{E}[W_n]$ for $n = 1, 2, \ldots 200$, using 10^6 simulation runs. For $n = 1, 2, 10, 50, 100, 200$ the simulated values and the difference between the steady-state mean waiting time and the simulated mean waiting times are given in Table 8.1. Obviously, the first customer entering the system

n	1	2	10	50	100	200	500	1000
$\mathbb{E}[W_n]$	0.000	0.486	1.985	3.466	3.778	3.880	3.887	3.887
$\mathbb{E}[W] - \mathbb{E}[W_n]$	3.889	3.403	1.904	0.423	0.111	0.009	0.002	0.001

Table 8.1: Simulated values for $\mathbb{E}[W_n]$ in the $M/M/1$ queue compared to the steady-state limit $\mathbb{E}[W]$, based on 10^6 runs.

Listing 8.29: More efficient simulation of the processor sharing queue.

```
def simulate(self, T):
    res = SimResults()                          # simulation results
    queue = array([])                           # empty queue
    t = 0                                       # current time
    nextArr = self.arrDist.rvs()                # next arrival time
    while t < T :                               # main loop
        oldQL = len(queue)
        nextDep = inf
        if len(queue) > 0:
            nextDep = queue[0]

        if nextArr < nextDep:                   # arrival event
            t = nextArr
            nextArr = t + self.arrDist.rvs()
            if oldQL > 0:           # change job completion times
                dep = t + self.servDist.rvs() * oldQL
                idx = searchsorted(queue, dep, side='left')
                queue = insert(queue, idx, dep)
                queue = t + (queue - t) * ((oldQL + 1) / oldQL)
            else:                   # system was empty
                dep = t + self.servDist.rvs()
                queue = insert(queue, 0, dep)
        else:                                   # departure event
            t = queue[0]                        # time of departure
            queue = delete(queue, 0)            # remove completion time
            queue = t + (queue - t) * ((oldQL - 1) / oldQL)
        res.registerQueueLength(t, oldQL)  # register queue length
    return res
```

experiences no waiting time at all. The mean waiting time of the second customer entering the system is 0.486. The 100^{th} customer experiences a waiting time that is 0.111 (so still almost 3%) less than the mean waiting time in steady state. This raises a few practical issues:

- Should you actually be interested in the steady-state mean waiting time? Suppose that this $M/M/1$ queue would be used to model a real-life situation with $\lambda = 0.7$ customers arriving per minute, and service times taking on average $1/\mu \approx 1.1$ minute. If customers arrive for eight hours per day, this corresponds to 480 customers per day. The *mean* waiting time of the first 480 customers is equal to 3.73 minutes, which is almost 10 seconds less than the steady-state mean waiting time.
- If indeed we are interested in the steady-state mean waiting time, how should we determine how long to simulate, and how to get rid of this underestimation of the steady-state mean waiting time due to the initial empty state of the simulation?

The first issue is obviously one that we cannot determine for you. We only want to stress that often there is no particular reason to study steady-state behaviour. The second issue will be addressed in the next section.

8.5 Warm-up effect and confidence intervals

In this section we continue the discussion on simulating the steady-state behaviour of stochastic processes. From Table 8.1 we can clearly conclude that we can approximate the mean waiting time in an $M/M/1$ queue with $\lambda = 0.7$ and $\mu = 0.9$ with $\mathbb{E}[W_n]$ for $n \geq 1000$. However, due to the fluctuations in the individual waiting times, we need a huge amount of runs to obtain a narrow confidence interval. It is much more efficient to use the *average* of multiple successive waiting times per run. Figure 8.9 suggests that

$$\mathbb{E}\left[\overline{W}_{n,N}\right] := \mathbb{E}\left[\frac{1}{N-n+1}\sum_{i=n}^{N} W_i\right], \qquad n = 1, 2, \ldots, N$$

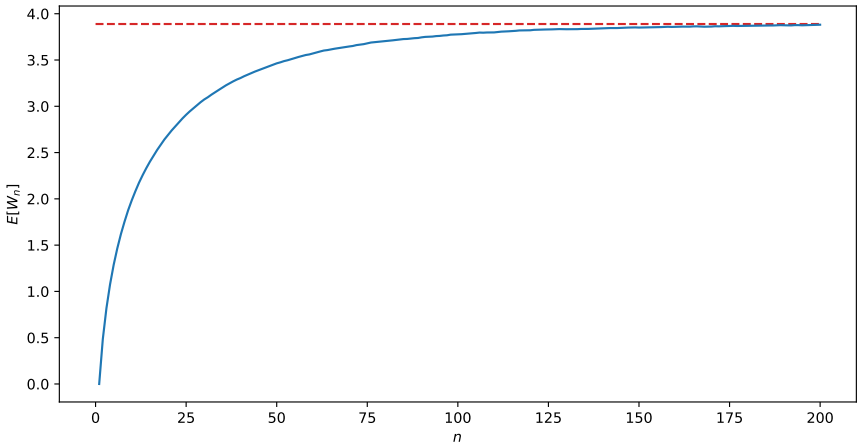


Figure 8.9: A plot of simulated values of $\mathbb{E}[W_n]$ for $n = 1, 2, \ldots, 200$. The simulated values are connected by a solid line. The theoretical steady-state limit for $n \to \infty$ is depicted by the dashed line.

is also a good approximation for $\mathbb{E}[W]$ for n and N sufficiently large. Clearly the variance of $\overline{W}_{n,N}$ is much smaller than the variance of W_N. Table 8.2 depicts the confidence intervals for $\mathbb{E}[W]$ based on $\mathbb{E}[W_N]$, $\mathbb{E}\left[\overline{W}_{1,1000}\right]$, $\mathbb{E}\left[\overline{W}_{200,1000}\right]$ (all 10^6 runs) and $\mathbb{E}\left[\overline{W}_{1,10^6}\right]$ (1000 runs). From this table we can conclude that $\mathbb{E}[W_N]$ gives an accurate estimate, but results in a relatively wide confidence interval. Using $\mathbb{E}\left[\overline{W}_{1,1000}\right]$ as an estimator results in a narrow confidence interval, the estimate is highly inaccurate. Phrased differently, the *sampling error* is small, but the *systematic error* caused by the warm-up effect is large. Therefore, a better estimator is $\mathbb{E}\left[\overline{W}_{200,1000}\right]$ which has all desirable properties. Determining the length of this so-called *warm-up* interval, or *burn-in* interval, can be done by investigating the transient behaviour of the system as discussed in the previous section.

Confidence intervals for $\mathbb{E}[W]$				
Based on	Lower bound	Upper bound	Width	Accuracy
$\mathbb{E}[W_{1000}]$	3.883	3.902	Wide	Good
$\mathbb{E}[\overline{W}_{1,1000}]$	3.809	3.813	Narrow	Bad
$\mathbb{E}[\overline{W}_{200,1000}]$	3.887	3.892	Narrow	Good
$\mathbb{E}[\overline{W}_{1,10^6}]$	3.888	3.892	Narrow	Good

Table 8.2: 95% confidence intervals for the steady-state mean waiting time, based on 10^6 runs (except the last interval, which is based on 1000 runs).

Note that the *variance* of the waiting time of an arbitrary customer, $\mathrm{Var}[W]$, just like $\mathbb{E}[W]$, can be determined by taking *one* single simulation run. If we have a single run with waiting times $w_1, w_2, \ldots, w_N$ (preferably after removing the first waiting times to eliminate a possible warm-up effect), we have for large N:

$$\mathbb{E}[W] \approx \frac{1}{N}\sum_{i=1}^{N} w_i, \qquad \mathrm{Var}[W] \approx \frac{1}{N}\sum_{i=1}^{N} w_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} w_i\right)^2.$$

Moreover, a (scaled) histogram of $w_1, w_2, \ldots, w_N$ gives a good impression of the density of the waiting-time distribution. Conclusion: you do *not* need to do multiple runs to obtain good estimates for the mean, variance, or even the distribution of the waiting time. The same is true for the queue length. However, if you want a confidence interval, you really need to do multiple runs, because the dependence of the successive waiting times violates the assumptions required to construct a confidence intervals. More about this in the next paragraph.

Confidence intervals. The confidence intervals are all constructed using independent simulation runs. A frequently made mistake is to use only *one* simulation run, yielding waiting times $w_1, w_2, \ldots, w_N$, and constructing a confidence interval from these N observations. The problem lies in the fact that the successive waiting times are dependent, als illustrated in the realisation of one single simulation run in Figure 8.10. Instead one should do *multiple* runs, say M runs. For run i,

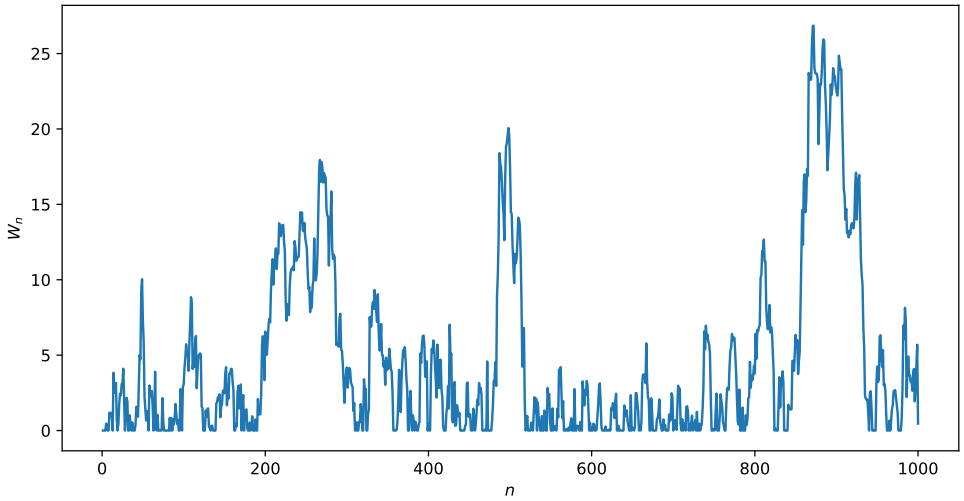


Figure 8.10: A plot of one sample path of simulated values of W_n for $n = 1, 2, \ldots, 1000$. The simulated values are connected by a solid line.

$i = 1, 2, \ldots, M$, determine the *mean* waiting time $\overline{w}_i = \frac{1}{N}\sum_{j=1}^{N} w_{i,j}$, where $w_{i,j}$ is the waiting time of the j-th customer in run i. Then use these M mean waiting times to construct a $100(1-\alpha)\%$ confidence interval using Equation (2.14). Be very careful here: the number n in (2.14) corresponds to the number of observations on which this confidence interval is based, which is equal to M (the number of runs). A frequently made mistake is that one takes N here, the length of a single run.

Batch means. A disadvantage of the aforementioned procedure to construct confidence intervals is that each separate simulation run needs to have a warm-up interval. The method of batch means states that instead of doing M simulation runs of $N + n$ waiting times (n is the number of removed observations due to the warm-up effect) it is also possible to to *one long* simulation run and, after deleting the first k observations, dividing this run into M sub-runs. Both methods result in M averages of N waiting times, but the first method requires $M(N + n)$ observations, whereas the second method requires only $MN + n$ observations. Although there is dependence between successive simulated waiting times, this dependence reduces quickly when the observations are further apart in the simulation run. Hence, the estimators $\mathbb{E}[\overline{W}_{n+1,n+N}], \mathbb{E}[\overline{W}_{n+N+1,n+2N}], \ldots \mathbb{E}[\overline{W}_{n+(M-1)N+1,n+MN}]$ can be regarded as mutually independent, making it possible to construct confidence intervals based on them.

An implementation in R of the batch means method for the $G/G/1$ queue can be found in Listing 8.30.

Listing 8.30: The batch means method applied to the $G/G/1$ queue.

```
1  def ciBatchMeans(M, N, k):
2      sim = simGG1Lindley2(arrDist, servDist, M * N + k)
3
4      # throw away the first k observations, and divide the rest into
5      # subruns of length N each
6      run = sim[k:(M * N + k)]
7      p = reshape(run, (M, N))
8      sample = mean(p, axis=1)  # take row means
```

```
 9       meanW = mean(sample)
10       varW = var(sample)
11       ci = meanW - 1.96*sqrt(varW/M), meanW + 1.96*sqrt(varW/M)
12       return ci
```

This code performs one simulation run of $200 + 10^8$ waiting times. With $M = 10^3$ and $N = 10^5$ the estimate for $\mathbb{E}[W]$ is 3.888, and a 95% confidence interval is $[3.880, 3.895]$. An interesting question is obviously what values for M and N should be used, given that MN remains constant, in this case 10^8. Note that the estimate for $\mathbb{E}[W]$ remains unchanged, regardless of how the long run is split into sub-runs. There is a trade-off between the number of sub-runs and their length. The value of M should not be too small, because that would result in wider confidence intervals; the value of N should not be too small because that would result in dependence between successive mean waiting times. Given that M and N are sufficiently large (at least 1000) it actually hardly matters how the values are chosen. They all result in more or less the same confidence intervals.

The regenerative method. The regenerative method is an intelligent method exploiting the special structure of a specific clas of stochastic models, requiring *no warm-up* interval. In the regenerative method, we again do only one simulation run but instead of dividing this simulation run into M equal-size sub-runs, we divide it into M *unequal-size* sub-runs. More specifically, the method makes use of the fact that many stochastic processes are *regenerative*. This means that there are random points in time, so-called regeneration points, such that the process starts afresh probabilistically at these time points. The behaviour of the process after the regeneration point is independent of and probabilistic identical to the behaviour of the process before the regeneration point. Now, in the regenerative method we start a new sub-run at time points that the system regenerates itself. In this way, the method provides a solution to the problems of how to start a simulation and how to deal with correlated output.

The disadvantage of this method is that it requires some knowledge about renewal theory to determine whether a process actually has regeneration points. It is beyond the scope of these lecture notes to discuss renewal theory, so we will only mention three examples of regenererative processes here:

- The $G/G/1$ queue. In the $G/G/1$ queue the moments that a customer arrives at an empty queue are regeneration points.
- On-off fluid model. In the two-machine production line of Section 7.4, there are two types of regeneration points. The first type of regeneration points are the moments that machine 1 is repaired and the buffer is empty. The second type of regeneration points are the moments that machine 1 breaks down and the buffer is completely filled.
- Periodic review inventory system. Consider a periodic review inventory system with independent, identically distributed demands per period and instantaneous delivery of replenishment orders. Demand that cannot be satisfied directly from stock is lost. The stock level is controlled by an (s, S) policy. For this system, the regeneration points are the moments just after an arrival of a replenishment order.

We shall illustrate the method using again the simulation of the long term average waiting time in the $G/G/1$ queue. Let N_i be the number of customers served in the i-th regeneration period (= sub-run) and let Y_i be the sum of the waiting times of the customers that are served in the i-th regeneration period, i.e.

$$Y_i = \sum_{j=1}^{N_i} W_{i,j},$$

where $W_{i,j}$ is the waiting time of the j-th customer served in the i-th regeneration period. The outcome of the i-th sub-run of our simulation is now the pair of random variables (Y_i, N_i) instead of only Y_i/N_i

as was the case in the previous methods. This is due to the fact that the number of customers served in the different sub-runs is not equal!

Now, the theory of regenerative processes tells us that the long-term average waiting time of customers can be calculated using information over only one regeneration period. More specifically,

$$\mathbb{E}[W] = \frac{\mathbb{E}[Y_i]}{\mathbb{E}[N_i]}.$$

Hence, an estimator for $\mathbb{E}[W]$ is given by

$$\overline{W} = \frac{\overline{Y}}{\overline{N}}.$$

For the construction of a confidence interval, we define the random variables

$$V_i = Y_i - \mathbb{E}[W]N_i. \qquad , i = 1, 2, \ldots, M.$$

The V_i's are independent, identically distributed random variables with mean 0 and variance

$$\sigma_V^2 = \sigma_Y^2 + \mathbb{E}[W]^2\sigma_N^2 - 2\mathbb{E}[W]\,\mathrm{Cov}(Y, N).$$

Using the central limit theorem for the V_i's it is straightforward to show that

$$\left[\overline{W} - \frac{z_{\alpha/2}\sigma_V}{\overline{N}\sqrt{M}}, \overline{W} + \frac{z_{\alpha/2}\sigma_V}{\overline{N}\sqrt{M}}\right]$$

is an approximate $(1-\alpha)100\%$ confidence interval for $\mathbb{E}[W]$. The unknown quantity σ_V^2 can be estimated by the sample variance

$$s_V^2 = s_Y^2 + \overline{W}^2 s_N^2 - 2\overline{W}s_{Y,N},$$

where

$$s_Y^2 = \frac{1}{M-1}\sum_{i=1}^{M}(Y_i - \overline{Y})^2,$$

$$s_N^2 = \frac{1}{M-1}\sum_{i=1}^{M}(N_i - \overline{N})^2,$$

$$s_{Y,N} = \frac{1}{M-1}\sum_{i=1}^{M}(Y_i - \overline{Y})(N_i - \overline{N}).$$

An implementation in Python of the regenerative method for the $G/G/1$ queue can be found in Listing 8.31.

This code performs one simulation run of 10^8 waiting times. In our experiment, the single run was divided into $M = 22,213,527$ sub-runs. The estimate for $\mathbb{E}[W]$ is 3.893, and a 95% confidence interval is $[3.885, 3.901]$.

To summarize, we have that the regenerative method has the advantages that successive observations are really independent and that you do not have to deal with the initialization effect. Disadvantages of the method are that not all systems have regeneration points and that hence the method is not generally applicable. Furthermore, visits of regeneration points can be very rare, especially for heavily-loaded systems.

Listing 8.31: The regenerative method applied to the $G/G/1$ queue.

```
def ciRegenerative(N):
    sim = simGG1Lindley2(arrDist, servDist, N)
    # Now we're going to split the simulation results vector every time we
    # encounter an empty system (i.e. a waiting time of zero)
    idx = where(sim == 0)[0]      # the positions of the zeros
    sa = split(sim, idx)          # split the list into sub-lists
    Yi = [sum(x) for x in sa]     # the sum of the waiting times in each sub-list
    Ni = [len(x) for x in sa]     # the number of waiting times in each sub-list
    M = len(Yi)                   # the number of sub-lists
    Yavg = mean(Yi)               # The average of the sums of the waiting times
    Navg = mean(Ni)       # the mean number of waiting times of the sub-lists
    Wavg = Yavg / Navg            # The overall mean waiting time

    cv = cov(Yi, Ni)[0,1]  # sample covariance is element at (0, 1) or (1, 0)
    sV2 = var(Yi) + Wavg**2 * var(Ni) - 2 * Wavg * cv
    print(sV2)
    ci = Wavg - 1.96 * sqrt(sV2/M)/Navg, Wavg + 1.96 * sqrt(sV2/M)/Navg
    return(ci)
```

Remark: A similar analysis is possible if we are dealing with a performance measure in continuous time (like queue length) instead of a performance measure in discrete time (like waiting time). For example, if we are interested in the long-term average number of customers in the system of the $G/G/1$ queue, then we could define

$$\begin{aligned} Y_i &= \int_{\alpha_i}^{\alpha_{i+1}} Q_t \; dt, \\ N_i &= \alpha_{i+1} - \alpha_i, \end{aligned}$$

where $\alpha_1, \alpha_2, \alpha_3, ...$ are the successive regeneration points and Q_t is the number of customers in the system at time t. Hence, N_i represents the length of the i-th regeneration period and Y_i represents the integrated queue length over the i-th regeneration period. Now, the construction of an estimator and a confidence interval is the same as before.

9

Process interaction approach with SimPy

SimPy is a process-based discrete-event simulation module based on standard Python. SimPy simulations are based on the discrete-event scheduling techniques discussed in the previous chapter, but the difference lies in the fact that SimPy is a framework that takes care of the underlying event scheduling. Instead, the programmer defines the *processes* that take place in the queueing model. This makes the code elegant, easier to understand, and much better suitable for network models, in particular when multiple types of resources are involved. Unfortunately, there are some disadvantages too: SimPy simulations are generally (slightly) slower than tailor-made implementations and, despite its flexibility, it may be difficult to implement advanced, nonstandard features (for example, different queueing/service disciplines).

In this chapter we provide an introduction to SimPy, re-implementing the examples from the previous chapter. A much more extensive tutorial is available at the SimPy homepage.

9.1 Example: An on-off fluid model

We start by revisiting the on-off fluid model from Section 7.4. This is a relatively simple process and the SimPy implementation does not differ much from the implementation in Listing 7.8. The major differences lie in the fact that we now have to put everything in the context of a SimPy environment and define the buffer process. The process alternates between up and down times, each implemented by invoking the `timeout(time)` function in the SimPy environment. After each up/down time, the buffer content is updated according to the formulas discussed in Section 7.4. Note that the main loop in this process seems to run forever, but the environment will make sure that the process is terminated after the specified run time

The main program initialises the SimPy environment, creates the buffer process, and runs this process for a maximum simulation length of 3 million time units. Since a process is not a function that returns a value (for example, the relevant performance measure), we use a global variable `th` to store the throughput. This variable is updated inside the process function. The complete source code is shown in Listing 9.1.

9.2 Example: the $G/G/c$ queue

A more interesting example to illustrate the advantages of the process interaction approach, compared to the discrete-event simulation in Section 8.3, is the multi-server queue from Example 8.1. Now there are two processes that need to be defined: the arrival process and the customer process.

Listing 9.1: SimPy implementation of the on-off fluid model.

```
from scipy import stats
from dist.Distribution import Distribution
import simpy

th = 0  # global variable for throughput

def simBuffer(env, lam, mu, r1, r2, K):
    global th       # use the GLOBAL variable 'th' for throughput
    b = 0           # current buffer content
    empty = 0       # total time that the buffer is empty
    upDist = Distribution(stats.expon(scale=1/lam))
    downDist = Distribution(stats.expon(scale=1/mu))
    while True :
        u = upDist.rvs()        # generate a random up time
        yield env.timeout(u)    # let the process delay for u time units
        b = min(b + u * (r1 - r2), K)   # update buffer level after up time
        th = r2 * (1 - empty/env.now)   # update the throughput
        d = downDist.rvs()      # generate down time
        yield env.timeout(d)    # delay for d time units
        b -= d * r2             # update buffer level after down time
        if b < 0 :              # buffer level dropped below zero
            empty -= b / r2
            b = 0
        th = r2 * (1 - empty/env.now)

env = simpy.Environment()   # create the SimPy environment
proc = simBuffer(env, 1/9, 1, 5, 4, 10)  # create the simulation process
env.process(proc)           # add the process to the simulation environment
env.run(until=3000000)      # let the environment run for 3000000 time units
print('Throughput = %f' % th)
```

Arrival process. Generates the arrivals of the customers. In this implementation, we have chosen to specify a fixed number of arrivals on beforehand ($n = 100000$). Each arrival spawns a Customer process. Between arrivals this process is delayed for a random interarrival time using the `timeout(time)` function in the SimPy environment. This implementation is typical for Poisson (or even renewal) arrival processes, which are very simple and easy to implement.

Customer process. The customer process starts by recording the current time (using `env.now`), which is used to determine the waiting time later. Next, a resource (i.e., one of the c servers) is requested. The request and waiting for an available server is handled automatically by the SimPy environment. As soon as a server becomes available, we record the waiting time, generate a random service time b, and delay this process for this time b. Upon service completion, because we used the `with ... as` construction, the SimPy environment will automatically release this server, such that it becomes available for the next waiting customer. See the SimPy documentation about the synchronisation issues that may occur if you do *not* use this construction, and manually release a resource instead. Because of these possible synchronisation issues, the `release` method has become obsolete and automatic release of the resource is recommended. An implementation in SimPy of this process is shown in Listing 9.2.

A major advantage of the process interaction approach is that we can focus on the processes themselves, without worrying about which events have to be scheduled at which time epochs. The service process is more or less self-organised. A disadvantage, is that when one wants to model a process that is different from the standard first-come-first-served policy, significantly more effort is required. Moreover, keeping track of *queue lengths* also requires a different approach. We refer to the SimPy manual (and Section 9.4) for more information on this topic. In the next example, we show how to simulate the queueing network with customer impatience of Section 8.3 with SimPy.

Listing 9.2: SimPy implementation of the $G/G/c$ queue.

```
def arrivalProcess(env, number):
    for i in range(number):              # generate the 'number' arrivals
        a = arrDist.rvs()                # generate a random interarrival time
        yield env.timeout(a)             # delay this process for a time units
        c = customerProcess(env, i)      # for each customer create a process.
        env.process(c)                   # start the process

def customerProcess(env, nr):
    arrivalTime = env.now
    with server.request() as req:        # request a server
        yield req                        # wait for the server to become available
        w = env.now - arrivalTime        # request is granted, waiting time ends
        waitingTimes[nr] = w             # store the waiting time
        b = servDist.rvs()               # generate the service time
        yield env.timeout(b)             # delay this customer process

n = 100000              # number of customers to simulate
waitingTimes = zeros(n) # list that will contain all the waiting times
lam = 2.4
mu = 1.0
c = 3
arrDist = Distribution(stats.expon(scale=1/lam))
servDist = Distribution(stats.expon(scale=1/mu))
env = simpy.Environment()
server = simpy.Resource(env, capacity=c)
env.process(arrivalProcess(env, n))
env.run()
meanW = mean(waitingTimes)
print('Mean waiting time: %f' % meanW)
```

9.3 Example: Customer impatience in a network of multiple queues in series

In this example we revisit the tandem queueing network from Example 8.2. The readers will notice that the SimPy implementation of this model is extremely elegant and the advantages of the process interaction approach certainly become clear in this example. We again follow the object oriented approach from Section 8.3. In fact, the Customer object is much simpler, not requiring the `moveTo` and `leaveSystem` functions. See Listing 9.3. The `NetworkSimResults` and `SimResults` classes are exactly the same as in the previous chapter.

Listing 9.3: SimPy tandem queueing network with impatience: Customer

```
class Customer :

    def __init__(self, location, systemArrivalTime, endPatienceTime):
        self.location = location
        self.systemArrivalTime = systemArrivalTime
        self.endPatienceTime = endPatienceTime
```

The main simulation (again) contains an arrival process and a customer process. However, due to the routing and the impatience, these processes become more complicated. The arrival process new creates actual customer entities, which also requires sampling their impatience. The customer process takes care of the waiting, the service, the routing, and the impatience. Note that a customer process ends when a customer is served, and a new process is started at the next queue. The process starts by scheduling two sub-processes in parallel:

1. waiting for an available server, and
2. wait for the customer's patience to run out.

By using the Python command

```
processResult = yield serverReq | endPatienceEvent
```

these environment will schedule both events (internally) and wait for the first one to take place, which is stored in the variable `processResult`. The other event is discarded.

If the server becomes available before the patience runs out:

- compute and register the waiting time,
- sample a random service time,
- start the following two processes in parallel (again): (1) service process and (2) impatience process. The result will be stored in the variable `processResult2`,
- If the service ends before the impatience runs out, depending on the location of the customer, either move the customer to the next queue and start a new customer process there, or register his sojourn time (and the customer leaves the system).
- If, however, the impatience runs out during the service, register the abandonment. The SimPy environment will discard the previously scheduled end-of-service event.

If the patience runs out while waiting for an available server:

- register the abandonment. The SimPy environment will discard the server request by this customer.

This can all be found in Listing 9.4. The last part of this program creates the interarrival-time and patience-time distribution, the service-time distributions, initialises the NetworkSimResults object, creates the resources, and starts the SimPy environment.

9.4 Monitoring resources

In the previous examples, we only recorded waiting times and/or sojourn times. Recording the queue length would be more difficult, because the whole queueing process is handled automatically by the resources (i.e., the servers) in SimPy. It is possible, however, to create a class called `MonitoredResource` that inherits from the standard SimPy Resource class. By overriding the methods that are invoked whenever an entity joins or leaves the queue, we can compute all queue length related performance measures in the regular way, in the `SimResults` object. These methods are called `request` and `release`, respectively. The number of *waiting* customers plus the number of customers in service is

```
1  ql = len(self.queue) + self.count
```

An example of how to extend the SimPy Resource class is shown in Listing 9.5.

Using such a MonitoredResource object is straightforward: in Listing 9.4 we should replace the line

```
1  server = simpy.Resource(env, capacity=c)
```

by

```
1  server = MonitoredResource(env, capacity=c)
```

Recording the waiting times can now also be done by the SimResults object inside the server. See Listing 9.5 for more details.

9.5 Example: The processor sharing queue

A slightly more complicated model to implement in SimPy, is the processor sharing queue from Example 8.4. On the one hand, there is no queueing involved, because all customers are taken into service immediately. On the other hand, the event times change upon each arrival or service completion, requiring all events to be rescheduled regularly. A standard resource in SimPy can serve only one customer at a time, so we need to be creative when implementing the processor sharing principle. In this case, the most obvious solution is to use a resource with an infinite capacity, like an

$M/G/\infty$ queue. To reschedule the events upon each arrival and departure, we need to *interrupt* all active service processes and restart them with an appropriately recomputed residual service time. To achieve this, we need to create a separate service process within the customer process. The reason why we need a separate service process, is because we need a process that can be interrupted (and restarted) without cancelling the overall customer process.

Arrival process. The arrival process is exactly the same as for the $G/G/c$ queue, as discussed in Listing 9.2.

Service process. The SimPy implementation of the service process is shown in Listing 9.6. This process must be constructed in such a way, that it is possible to interrupt the process and create a new end-of-service event. For this reason, there is a continuous loop (`while True`) that ends (`break`) when the (final part of the) service is completed successfully. Upon each interruption, whenever a new customer arrives or another customer leaves the system, the remaining service requirement is computed and a new iteration of the while-loop starts.

Customer process. The customer process is not too different, compared to the $G/G/c$ queue. There are two main differences: first, as mentioned before, the service is now really a separate process (`env.process`) so that it can be interrupted. To interrupt the service processes of all customer present in the system, we create a list (`allServiceProcesses`), which is a deque data structure, that contains all the running service processes. Whenever a service process starts, it is appended to this list, and it is immediately removed when the process ends. The interruptions take place whenever a customer arrives (i.e., at the start of the customer process) and after their departure (at the end of the customer process). Listing 9.7 shows the SimPy implementation.

9.6 Example: The matching queue

Implementing the matching queue from Example 8.3 in SimPy requires a different approach. In this example, one of the *disadvantages* of a system like SimPy becomes apparent: non-standard cases that are not implemented by default in SimPy, might be difficult to implement. The complication is the following: in SimPy, one has to determine exactly which resource is requested. In case of multiple servers, it is possible to increase the capacity of a certain resource (as illustrated in the $G/G/c$ example), but this automatically implies that all servers are considered to be identical copies. In case there are different servers, things become more complicated, and when one needs more control over which server should be requested, this may not be possible at all using SimPy resources.

Concretely, there are two issues:

- The servers are not identical (they have different service-time distributions for different customer types) and not all servers are compatible with an arriving customer. This means that so we cannot specify on beforehand exactly *which* server we want.
- Since the system is no longer *work conserving*, i.e., there might be idle servers despite the fact that there are (incompatible) waiting customers, we also need to specify a policy which server to match with a newly arriving customer. In this example, we use the Assign to Longest Idle Server (ALIS) policy. This is a policy that we have to implement ourselves, it is not readily available in SimPy.

To overcome these two issues, we have decided *not* to use SimPy resources to represent the servers, but to create Server objects manually. As a consequence, the implementation looks much more similar to the implementation in Example 8.3, without SimPy. Indeed, we use the same Customer,

Server, and SimResults classes as in the original example. Still, the fact that we can use SimPy processes, makes it slightly more elegant. Besides an arrival process and a customer process, we now also need a server process. These three processes are discussed in more detail below.

Arrival process. The arrival process is not very different from what we have seen before. Note that we create a Customer object, with a specified customer type, which we pass as a parameter to the customer process. We start one arrival process for *each* customer type, so in case of *I* customer types, we have *I* arrival processes running in parallel.

Customer process. In contrast to the arrival process, the customer process is completely different from what we have seen before in our SimPy examples. We will discuss all the steps below and then provide the complete source code in Listing 9.9:

- We check the current time and register the current queue length. Note that registering the queue length now has to be done manually, since we do not use (monitored) resources in this example.
- We try to find a compatible server in the list of idle servers. This code is *exactly* the same as in Example 8.3. By having our own implementation of the idle server queue, we can ensure the ALIS policy.
- If no compatible server is found, the customer is added to the queue (which, again, we implement ourselves instead of using the queue inside a SimPy resource). Note that, in this case, this customer process ends! This does not mean that the customer has left the system, but it is simply waiting in its queue for another process to pick it up. This will be the Server process, which we discuss later.
- If a compatible server is found, we register the customer's waiting time (which is zero) and the server's idle time; we then sample a random service time and delay the process for this amount of time. After this service time, the server becomes available and needs to search for a compatible customer in the queue. This is implemented in the Server process.

Server process. the server process ensures that a server that just finished serving a customer, looks for a compatible customer in the queue and (if found) starts a new service. Since we have to implement this manually, this code is not too different from Example 8.3, but with the typical SimPy process timeouts. The astute reader will also notice that the code is similar to the customer process, but then with the roles of the sever and the customer reversed. We will discuss all the steps below and then provide the complete source code in Listing 9.10:

- We check the current time and register the current queue length.
- We try to find a compatible customer in the queue of waiting customers.
- If no compatible customer is found, the server is added to the idle server queue and this server process ends. At some point, the server will be matched with a compatible arriving customer inside a Customer process.
- If a compatible customer is found, we register the customer's waiting time; we then sample a random service time and delay the process for this amount of time. After this service time, the server becomes available and needs to search for a compatible customer in the queue. This is done by starting a new Server process for this server.

Listing 9.4: SimPy tandem queueing network with impatience: Main simulation

```
import simpy
from scipy import stats
from ch9.simpynetwork.Customer import Customer
from ch9.simpynetwork.NetworkSimResults import NetworkSimResults
from dist.Distribution import Distribution

def arrivalProcess(env, number):
    while results.nS < number:
        a = arrDist.rvs()       # generate a random interarrival time
        yield env.timeout(a)    # delay this process for this interarrival time
        endPatienceTime = env.now + patienceDist.rvs() # generate the patience
        cust = Customer(0, env.now, endPatienceTime) # create the customer
        c = customerProcess(env, cust)  # create a customer process
        env.process(c)          # start this process in the SimPy environment

def customerProcess(env, cust):
    arrivalTime = env.now           # arrival time of the customer at this station
    serverNr = cust.location        # current station number
    # for a Resource, always use this 'with ... as' construction!
    with servers[serverNr].request() as serverReq :   # request a server
        patience = cust.endPatienceTime - env.now # REMAINING patience
        endPatienceEvent = env.timeout(patience)
        processResult = yield serverReq | endPatienceEvent
        if serverReq  in processResult :   # the event = serverReq
            w = env.now - arrivalTime   # compute the waiting time
            results.registerWaitingTime(serverNr, w) # store the waiting time
            serviceTime = servDist[serverNr].rvs() # generate service time
            patience = cust.endPatienceTime - env.now # REMAINING patience
            serviceEvent = env.timeout(serviceTime) # create end-of-service event
            patienceEvent = env.timeout(patience) # create end-of-patience event
            processResult2 = yield serviceEvent | patienceEvent
            if serviceEvent in processResult2 :
                if serverNr < len(servers) - 1: # NOT in the last station
                    cust.location += 1 # move the customer to the next station
                    c = customerProcess(env, cust) # new customer process
                    env.process(c) # start this process
                else : # customer was in last station and leaves the system
                    results.registerSojournTime(env.now - cust.systemArrivalTime)
            else : # impatience event took place before the end-of-service event
                results.registerAbandonment(serverNr) # register the abandonment
        else : # impatience event took place before a server became available
            results.registerAbandonment(serverNr) # register the abandonment

n = 50000  # number of non-abandoning customers to simulate
arrDist = Distribution(stats.expon(scale=1/0.9)) # arrival distribution
mus = [1.0, 0.5, 0.25] # rates of the service-time distributions
nrStations = len(mus) # number of stations (= 3)
servDist = [Distribution(stats.expon(scale=1/mu)) for mu in mus]
patienceDist = Distribution(stats.expon(scale=1/0.1))
nrServers = [1, 3, 2] # number of servers of each station
results = NetworkSimResults(nrStations, n) # simulation results object
# Setup and start the simulation
env = simpy.Environment() # create the SimPy environment
servers = [None]*nrStations # create the resources
for i in range(nrStations):
    servers[i] = simpy.Resource(env, capacity=nrServers[i])
env.process(arrivalProcess(env, n)) # create the arrival process
env.run() # start the environment (with all the processes)
print(results)
```

Listing 9.5: A monitored resource in SimPy

```
import simpy

from ch9.simpynetwork.SimResults import SimResults

class MonitoredResource(simpy.Resource):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.simResults = SimResults()

    def request(self, *args, **kwargs): # join queue
        self.updateQL()
        return super().request(*args, **kwargs)

    def release(self, *args, **kwargs): # service completion
        self.updateQL()
        return super().release(*args, **kwargs)

    def updateQL(self):
        ql = len(self.queue) + self.count
        self.simResults.registerQueueLength(self._env.now, ql)
```

Listing 9.6: The service process in the SimPy implementation of the processor sharing queue.

```
def serviceProcess(env, b):
    while True:
        q = server.count                # number of customers
        t = env.now                     # current time
        try:
            yield env.timeout(b * q)    # delay this service process
            break                       # abort while loop, service finished
        except simpy.Interrupt:         # service interrupted
            b = b - (env.now - t)/q     # remaining service required
```

Listing 9.7: The customer process in the SimPy implementation of the processor sharing queue.

```
def customerProcess(env, nr):
    arrivalTime = env.now
    for p in allServiceProcesses:       # interrupt all other processes
        p.interrupt()
    with server.request() as req:       # request a server
        yield req                       # wait for the server to become available
        b = servDist.rvs()              # generate the service time
        serviceProc = env.process(serviceProcess(env, b)) # create service proc.
        allServiceProcesses.append(serviceProc) # add service process to list
        yield serviceProc               # start (and wait for) service process
        allServiceProcesses.remove(serviceProc) # remove process from list
    s = env.now - arrivalTime           # server is released, sojourn time ends
    server.simResults.registerSojournTime(s)    # store the sojourn time
    for p in allServiceProcesses:       # because the customer leaves, interrupt
        p.interrupt()                   # all other processes
```

Listing 9.8: The arrival process in the SimPy implementation of the matching queue.

```
def arrivalProcess(env, custType):
    while(True):
        a = arrDists[custType].rvs()    # generate a random interarrival time
        yield env.timeout(a)            # delay this process for a time units
        c = Customer(env.now, custType) # create a customer object
        cp = customerProcess(env, c)    # for each customer create a process.
        env.process(cp)                 # start the customer process
```

Listing 9.9: The customer process in the SimPy implementation of the matching queue.

```
def customerProcess(env, c):
    t = env.now                                 # current time
    results.registerQueueLength(t, len(q))      # register the total queue length
    # try to find compatible server
    s = None
    for i in range(len(idleServers)):           # Find compatible idle server
        s = idleServers[i]
        if s.compatible[c.type]:                # s is compatible with c
            del idleServers[i]                  # remove s from idle servers queue
            break
        else:                                   # s is not compatible with c
            s = None
    if s == None:                               # No compatible server found
        q.append(c)   # add customer c to the queue, wait for a server
    else:             # Compatible server found
        results.registerWaitingTime(t, c)
        results.registerIdleTime(t, s)
        serv = s.serviceTimeDists[c.type].rvs() # sample random service time
        yield env.timeout(serv)
        ip = idleServerProcess(env, s)  # server becomes available, start a
        env.process(ip)                 # new server process for this server
```

Listing 9.10: The server process in the SimPy implementation of the matching queue.

```
def serverProcess(env, s):
    t = env.now
    results.registerQueueLength(t, len(q))  # register the total queue length
    cust = None
    for i in range(len(q)):                 # Find compatible customer
        cust = q[i]
        if s.compatible[cust.type]:         # s is compatible with cust
            del q[i]                        # remove cust from queue
            break
        else:                               # s is not compatible with cust
            cust = None
    if cust == None:                        # No compatible customer found
        idleServers.append(s)         # add server s to the idle servers queue
        s.startIdleTime = t           # set start of idle time to current time t
    else:
        results.registerWaitingTime(t, cust)
        serv = s.serviceTimeDists[cust.type].rvs()
        yield env.timeout(serv)
        ip = serverProcess(env, s)          # start a new server process
        env.process(ip)                     # for this server
```

Part III

Applications

10

Epidemics

Epidemiology studies the outbreak of a virus and in particular the event that it turns into an *epidemic*. We speak of an epidemic when the number of infected people substantially exceeds the normal outbreak size. Various mathematical models, deterministic and stochastic, have been developed to describe the process dynamics of the outbreak of a virus. In this chapter we study the simulation of stochastic epidemic models. In this chapter we discuss so-called SIR models, which divides the population into Susceptible, Infected, and Recovered individuals. At the start of an outbreak, all people are susceptible and one (or a few) are infected. Infected people may transfer the disease to other susceptibles until they themselves become recovered. The "recovered" state is sometimes called "removed", indicating a person who no longer possesses the virus, either because he has recovered and become immune, or possibly because he is deceased. The total population size in the SIR model remains constant. We first discuss a discrete-time SIR model called the Reed-Frost model.

10.1 Discrete-time models

10.1.1 The Reed-Frost model

The Reed-Frost model is a discrete-time stochastic model, where the infection time of an individual is assumed to be exactly one time step. This means that each infected individual will be recovered in the next time step. Before an infected individual recovers, he may infect each of the susceptibles independently with probability p. The number of susceptible, infected, and recovered individuals at time n is denoted by S_n, I_n, and R_n respectively. The total population size remains constant, $S_n + I_n + R_n = N$ for all $n = 0, 1, 2, \dots$. The initial population state at time $n = 0$ is often taken to be $S_n = N - 1$, $I_n = 1$, and $R_n = 0$. Simulating the Reed-Frost model involves an iterative scheme where each step determines, for each susceptible, whether he will be infected by any of the infected individuals. Note that, at time n, the probability that he will be infected at time $n + 1$ is equal to $1 - (1 - p)^{I_n}$.

Simulation. Input: (S_0, I_0, R_0) and the probability p. Let $N = S_0 + I_0 + R_0$ be the total population size.

- Set $n = 0$
- While $I_n > 0$,
  - $R_{n+1} = R_n + I_n$
  - For each individual $i = 1, 2, \dots, S_n$ generate a random variate k_i from a binomial distribution with parameters I_n and p. This number k_i represents the number of infected people that

transfer the virus to susceptible individual i.
- $I_{n+1} = \sum_{i=1}^{S_n} \mathbf{1}_{\{k_i>0\}}$
- $S_{n+1} = N - I_{n+1} - R_{i+1}$
- $n = n + 1$

End while.

Note that the simulation ends at the moment that no individuals from the population are infected, which will always happen at some point. The length of the epidemic is the final value of n. An implementation in Python is given in Listing 10.1. A plot depicting the mean values for S_n, I_n, and R_n for a population of $N = 1000$ with one infected individual at time $n = 0$ and $p = 0.002$ is shown in Figure 10.1. Another interesting simulation result is the probability distribution of the size of an outbreak. Since every infected individual becomes recovered, this is the number of recovered individuals at the end of each simulation run. A histogram of the size of an outbreak is shown in Figure 10.2. This figure gives nice insights, for example that an outbreak either stays small or becomes an huge epidemic. When putting the threshold (arbitrarily) at 200, simulation shows that the outbreak stays small with probability approximately 0.2. The mean outbreak size *given that* it stays small is 1.65. However, given that it becomes an epidemic (with probability 0.8), the mean outbreak size is 796!

Listing 10.1: Source code used to simulate the Reed Frost model.

```
from collections import deque
from numpy.ma.core import zeros, array, mean
from scipy import stats
import matplotlib.pyplot as plt

def simReedFrost(N, i, r, p, minNrSteps=1):
    s = N - i - r
    sList = deque([s])
    iList = deque([i])
    rList = deque([r])
    while i > 0 or len(sList) < minNrSteps:
        r += i # Each infected person will become resistant
        i = sum(stats.binom(i, p).rvs(s) > 0)
        s = N - i - r
        sList.append(s)
        iList.append(i)
        rList.append(r)
    return (sList,iList,rList)

# Multiple runs
def simMultipleRuns(N, i, r, p, nrSteps, nrRuns):
    meanPath = array([zeros(nrSteps),zeros(nrSteps),zeros(nrSteps)])
    distNrRecovered = zeros(nrRuns)
    for j in range(nrRuns):
        samplePath = simReedFrost(N, i, r, p, nrSteps)
        for i in range(3) :
            meanPath[i] += array(samplePath[i])[0:nrSteps]/nrRuns
        # the last element of this list is the size of the outbreak:
        distNrRecovered[j] = samplePath[2][-1]
    return (meanPath,distNrRecovered)

nrRuns = 1000
nrSteps = 21
n = 1000
i = 1
r = 0
p = 0.002

meanPath,distRec = simMultipleRuns(n, i, r, p, nrSteps, nrRuns)
ss,ii,rr = meanPath
plt.figure()
plt.plot(ss,'b-')
plt.plot(ii,'r-')
plt.plot(rr,'g-')
plt.legend(['Susceptibles', 'Infected', 'Recovered'])

plt.figure()
plt.hist(distRec, bins=50, rwidth=0.8, density=True)
```

```
49  plt.show()
50
51  print(mean(distRec < 200))
52  print(mean(distRec[distRec < 200]))
53  print(mean(distRec > 200))
54  print(mean(distRec[distRec > 200]))
```

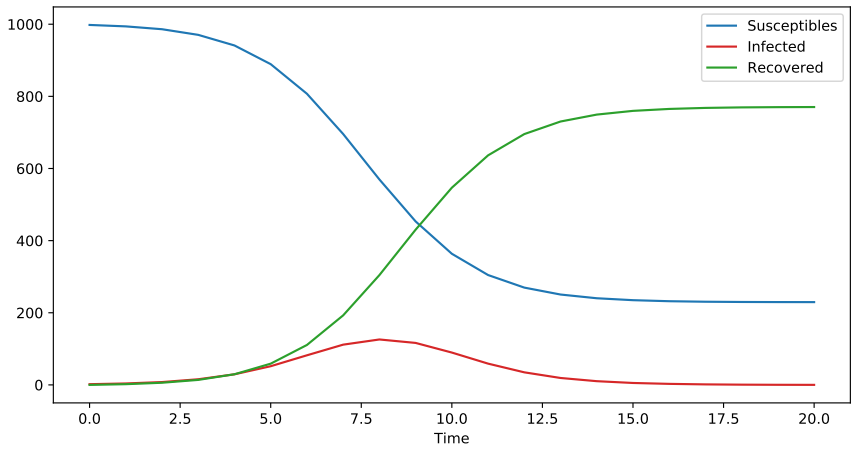


Figure 10.1: Mean simulated values for a Reed-Frost model with $N = 1000$ and $p = 0.002$, based on 10,000 simulation runs.

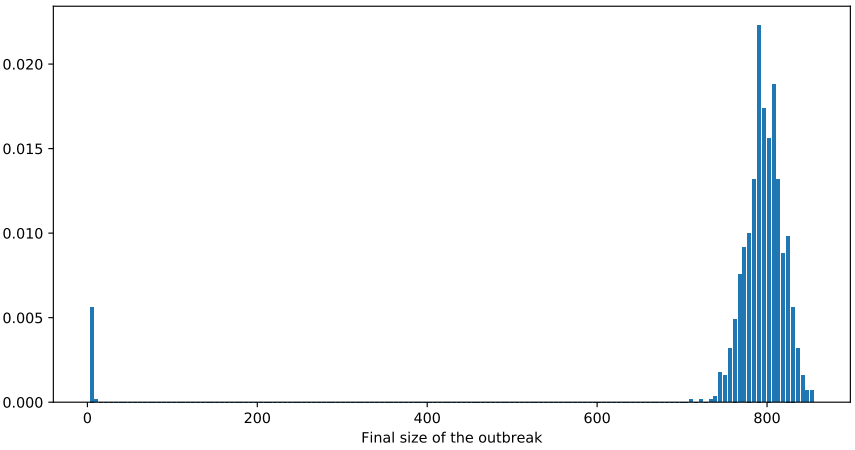


Figure 10.2: The distribution of the size of an outbreak of a Reed-Frost model with $N = 1000$ and $p = 0.002$, based on 10,000 simulation runs.

10.1.2 A spatial SIR model

A disadvantage of the Reed-Frost model is that it does not take into account spatial effects. The model discussed in this section is natural extension of the Reed-Frost model, with the spread of the virus taking place on a graph. The input is a graph $G = \{V, E\}$ of vertices and edges. Each vertex represents an individual and an edge between two vertices represents a connection between two individuals. The main assumption is that only connected individuals can infect each other, with

probability p. This affects the simulation, due to the fact that it is now required to keep track of the state of *each* individual, not just the number of individuals in each group.

Creating a random graph. In principle, only the nodes and the connections are relevant. They can be created by generating a random connectivity matrix M, with elements m_{ij} Bernoulli distributed. However, in this example we have chosen to generate points random in a $[0,1] \times [0,1]$ grid, with uniformly distributed x and y coordinates. Two points are connected if the distance between these points is less than a specified radius R. The code can be found in Listing 10.2.

Listing 10.2: Source code used to generate a random graph.

```
def generateConnectivityGraph(n, radius):
    # create connectivity matrix,
    # based on points closer to each other than distance 'radius'
    # This is called a geometric random graph
    x = random.uniform(0, 1, n)
    y = random.uniform(0, 1, n)
    mm = zeros((n, n))
    for i in range(n) :
        d = sqrt((x[i]-x)**2 + (y[i]-y)**2)
        mm[i] = (d < radius) * (d > 0)
    return mm
```

Creating a random initial state. We have concluded that the state of the graph is an N-dimensional vector X_0 with elements $x_{0,i} \in \{S, I, R\}$. Given the desired initial number of infected (I_0) and resistant (R_0) individuals, we use Monte Carlo simulation to generate a random initial state. The principle is simple: we create a random permutation of the numbers $1, 2, \ldots, N$. Without loss of generality, we can make the first I_0 items in this permutation infected, and the next R_0 individuals resistant. The Python code is given in Listing 10.3.

Listing 10.3: Source code used to generate a random initial state.

```
def generateRandomInitialState(nrTotal, nrInfected, nrResistant):
    nrSusceptible = nrTotal - nrInfected - nrResistant
    state = append(array([ReedFrostGraphModel.SUSCEPTIBLE]*nrSusceptible),
    [ReedFrostGraphModel.INFECTED]*nrInfected)
    state = append(state, [ReedFrostGraphModel.RESISTANT]*nrResistant)
    # create a random permutation of this state
    random.shuffle(state)
    return(state)
```

Simulation. Input: the graph G, the initial state X_0 (see previous paragraph), and the probability p. Let I_0 denote the number of items in X_0 equal to 'I' (infected).

- Set $n = 0$
- While $I_n > 0$,
  - First copy the state X_n: $X_{n+1} = X_n$.
  - All infected individuals in step n become resistant in step $n+1$:
    For all i with $x_{n,i} = I$ set $X_{n+1,i} = R$. Update the number of recovered, $R_{n+1} = R_n + I_n$, and reset the number of infected, $I_{n+1} = 0$.
  - For each susceptible individual i in X_n:
    - Determine all of his neighbours and count those who are infected, say k.
    - Determine the number of infected neighbours that will transfer the virus to him. The probability of this happening is p per infected neighbour, so the desired number is obtained by generating a random variate from a binomial(k, p) distribution.
    - If this number is greater than 0, the virus is transferred and the susceptible individual becomes infected: $x_{n+1,i} = I$ and $I_{n+1} = I_{n+1} + 1$. Otherwise the individual remains susceptible.

End for.
- Update the number of susceptibles: $S_{n+1} = N - I_{n+1} - R_{n+1}$.

End while.

An implementation in Python is given in Listing 10.4. Some plots of a random graph and the spread of a virus over this graph with $N = 250$ and radius 0.15 are given in Figure 10.3. We have used the NetworkX library to plot these graphs in Python, mainly because it is compatible with Matplotlib (used for the plots in these lecture notes) and it is easy to install on any operating system.

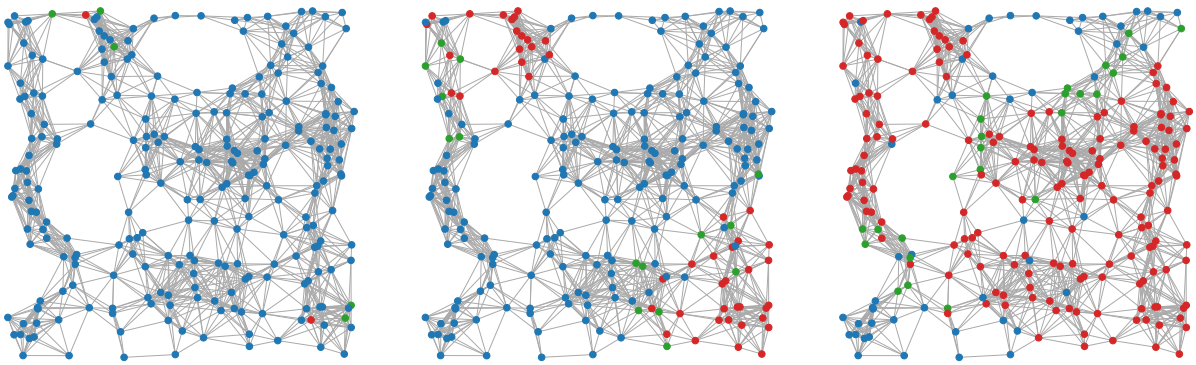
Listing 10.4: Source code used to simulate the discrete-time SIR model on a graph.

```
class ReedFrostGraphModel :

    SUSCEPTIBLE = 0
    INFECTED = 1
    RESISTANT = 2

    def __init__(self, connectivityMatrix=None):
        self.connectivityMatrix = connectivityMatrix

    def simulate(self, initialState, p):
        results = SimResultsReedFrost(initialState)
        state = array(initialState)
        nrInfected = sum(state == self.INFECTED)
        nrTotal = len(state)
        while nrInfected > 0 :
            # copy the last state
            nextState = state.copy()  # do not modify the original list
            # Each resistant person will stay resistant
            # Each infected person will become resistant
            nextState[state == self.INFECTED] = self.RESISTANT
            # Each of the susceptibles MIGHT get infected
            susc = state == self.SUSCEPTIBLE
            nrInfected = 0
            for i in range(nrTotal) :
                if susc[i] :
                    nrInfectedNeighbours = int(sum(self.connectivityMatrix[i]
                    * (state == self.INFECTED)))
                    # sample from binomial distribution
                    if (stats.binom(nrInfectedNeighbours, p).rvs() > 0) :
                        nextState[i] = self.INFECTED
                        nrInfected += 1
            results.addState(nextState)
            state = nextState
        return(results)
```



$n = 1$ $n = 5$ $n = 10$

Figure 10.3: Outbreak of a virus on a graph of 250 nodes, starting with two infected individuals.

10.2 Continuous-time models

In these lecture notes we do not discuss the simulation of continuous-time models. Instead, we just give a brief illustration of models encountered in the literature, which generally can be distinguished into two types. The dynamics are described either through stochastic differential equations, or using discrete jump Markov processes. Interestingly, the latter has mostly applications in chemistry. Chemical reaction systems with a low to moderate number of molecules are typically modelled as discrete jump Markov processes. These systems are oftentimes simulated with methods that produce statistically exact sample paths such as the *Gillespie algorithm* or the *next reaction method*. The Gillespie algorithm is basically a discrete-event simulation. In each step, first the time until the next event is simulated. It is argued that in chemically reacting systems the time until the next event is exponentially distributed. This property is sometimes called the fundamental premise of chemical kinetics [And07]. The time until the next event is the minimum of all possible events that could take place, all of which are assumed to be exponentially distributed. Hence, the minimum is exponentially distributed as stated by the aforementioned property. A discrete event simulation simply consists of scheduling the next event after an exponentially distributed time, and randomly generating the type of the event according to the probabilities on each event taking place.

11

Interacting particle systems

Interacting particle systems are continuous-time Markov jump processes describing the collective behaviour of stochastically interacting components, often taking place on a graph or grid. We illustrate the concept by giving an example first.

Example 11.1 (The voter model)**.** The most typical example of an interacting particle system is the so-called *Voter model* where the nodes represent voters having an opinion (0 or 1). At random epochs in time, a random individual may decide to change his opinion depending on those of his neighbours. In the Majority Vote Model random individuals change to the opinion of the majority of their neighbours, as illustrated in Figure 11.1. In the Threshold Voter Model individuals change their opinions by flipping a coin, as long as there are at least one 0 and one 1 among their neighbours' opinions. An interesting question is whether global consensus will be reached in the long term and, if so, whether it will be 0 or 1. An implementation in Python of the voter model is given in Listing 11.1.

Listing 11.1: Simulation of the Voter model.

```
class VoterModel :

    def __init__(self, connectivityMatrix) :
        self.connectivityMatrix = connectivityMatrix

    def simulate(self, initialState, nrSteps):
        results = SimResultsVoterModel(initialState)
        n = len(initialState)  # number of people
        state = array(initialState)
        for _ in range(nrSteps) :
            # copy the last state
            nextState = state.copy()  # create a copy
            # select a random person from 0, ..., n-1
            selectedIndex = rng.integers(0, n)
            neighbours = self.connectivityMatrix[selectedIndex].astype(int)
            nrNeighbours = sum(neighbours)
            nrVotes1 = sum(state[neighbours])
            if nrVotes1 > nrNeighbours / 2 :
                # majority votes "1"
                nextState[selectedIndex] = 1
            elif nrVotes1 < nrNeighbours/2 :
                # majority votes "0"
                nextState[selectedIndex] = 0
            else :
                # tie
                nextState[selectedIndex] = rng.integers(0, 2)  # 0 or 1
            results.addState(nextState)
            state = nextState
        return(results)
```

One of the most relevant interacting particle systems is the Ising model, which is a mathematical model used in statistical mechanics to describe the phenomenon of ferromagnetism, the thermodynamics of gases, and neuron activity. This model is described in Section 11.1. Another system which

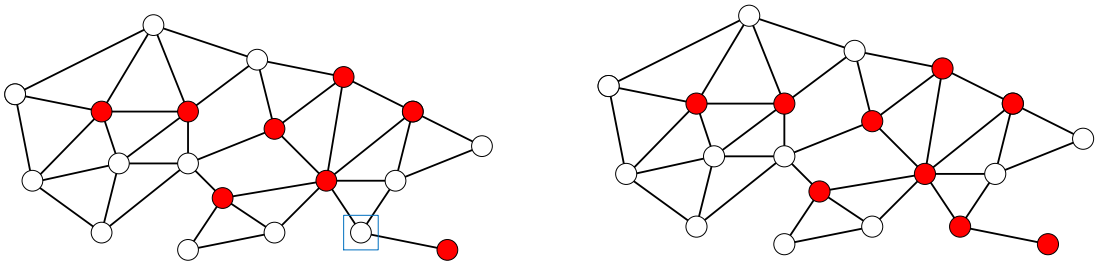


Figure 11.1: A graphical illustration of the voter model. The highlighted white dot, representing a voter, in the leftmost figure will change its opinion from 'white' to 'red', in accordance with the majority of its neighbours.

can be described as an interacting particle system, is a wireless network in which nodes represent devices trying to send messages. Simultaneous transmissions within a certain broadcast range might cause interference resulting in packet loss. For this reason, protocols have been developed that avoid packet loss by blocking nodes in the vicinity of another node that is already broadcasting. This example is described in Section 11.2. In fact, the spacial epidemics model described in the previous chapter, Section 10.1.2, can also be regarded as an interacting particle system. There interactions take place through infections of susceptible individuals.

11.1 The Ising model

The Ising model, named after the physicist Ernst Ising, is a mathematical model of ferromagnetism in statistical mechanics. The model consists of discrete variables representing magnetic dipole moments of atomic spins that can be in one of two states (spin up or down, represented in the mathematical model as $+1$ or -1). The spins are arranged in a graph, usually a lattice, allowing each spin to interact with its neighbours. One of the most interesting aspects of ferromagnetism is the so-called *phase transition* that takes place when the temperature rises beyond a certain threshold value, called the *Curie temperature*. The two-dimensional square-lattice Ising model is one of the simplest statistical models to show this phase transition.

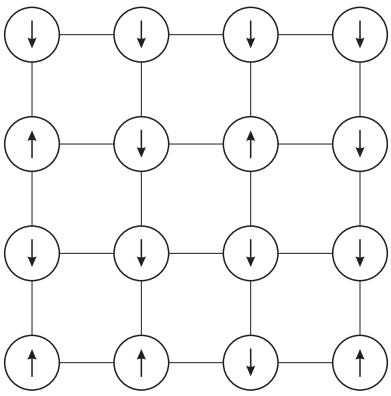


Figure 11.2: Spins on a lattice in the Ising model.

In this chapter we consider two-dimensional Ising magnets, which are represented as an $N \times N$ lattice of spins, each of which have a magnetic moment of $\pm\mu$. See Figure 11.2 for an example. The configuration of an Ising magnet is the list of spin variables $s_1, s_2, \dots, s_M$, where $M = N^2$. Let $\nu = (s_1, s_2, \dots, s_M)$ denote a configuration for a system with $s_i \in \{-1, 1\}$ for $i = 1, 2, \dots, M$. Note that there are 2^M different configurations. In order to determine the transition rates, we need to compute

the energy of the system in a configuration ν,

$$\begin{aligned} E_\nu &= -\mu H \sum_{i=1}^{M} s_i - J \sum_{<i,j>}^{M} s_i s_j \\ &= -\mu H \sum_{i=1}^{M} s_i - J \sum_{i=1}^{M} \sum_{j=i+1}^{M} s_i s_j \mathbf{1}_{\{i,j \text{ are neighbours}\}}, \end{aligned} \tag{11.1}$$

where the sum $< i, j >$ is taken over all pairs of neighbours, each pair being counted only once. In general each spin has four neighbours (left, right, up, and down), but at the boundaries a spin variable may have only two or three neighbours. In (11.1) J denotes the interaction strength between nearest neighbours. We will assume that $J = 1$, but in general a positive value of J indicates that it is energetically favourable for neighbouring spins to have the same sign ($\uparrow$ or $\downarrow$) (ferromagnetic interaction), and a negative value results in antiferromagnetic interaction. The parameter H is an external magnetic field, which we assume to be zero in our simulations, thereby considerably simplifying the expression for E_ν.

Due to the huge number of possible states, we use Monte Carlo simulation to generate trajectories, i.e. chronological sequences of configurations for a system. Most of the 2^M states have such high energy that their occurrence is negligible, and accordingly, will not be sampled in our Monte Carlo simulation. The most commonly used method is the Metropolis algorithm, which performs a discrete-time simulation. In each step, a random spin site s_i is selected (each with probability $1/M$). Next, the change in energy is computed that would result from flipping s_i. This change in energy is denoted by $\Delta E_{\nu,\nu'}$, where ν is the original configuration, and ν' is the configuration resulting from flipping. From Equation (11.1), and the fact that the only change between ν and ν' is in s_i, which has either changed from -1 to 1 or vice versa, it follows that

$$\Delta E_{\nu,\nu'} = 2 s_i (s_{i,L} + s_{i,R} + s_{i,U} + s_{i,D}),$$

where s_i corresponds to the value *before* the spin, and $s_{i,L}, s_{i,R}, s_{i,U}, s_{i,D}$ denote the four neighbours (left, right, up, down) in the grid. If $\Delta E_{\nu,\nu'}$ is negative, i.e. the energy of the new configuration is less than the energy of the original configuration, ν' will always be accepted as new configuration. If $\Delta E_{\nu,\nu'} \geq 0$, the flip is only accepted with a certain probability, namely $e^{-\beta \Delta E_{\nu,\nu'}}$. Summarising, the probability of acceptance a transition from configuration ν to ν', denoted by $A_\beta(\nu, \nu')$, is equal to:

$$A_\beta(\nu, \nu') = \begin{cases} 1 & \text{if } \Delta E_{\nu,\nu'} < 0 \\ e^{-\beta \Delta E_{\nu,\nu'}} & \text{otherwise.} \end{cases}$$

An implementation in Python of the Metropolis algorithm is given in Listing 11.2.

Listing 11.2: Simulation of the Ising model using the Metropolis algorithm.

```
1  rng = random.default_rng()    # The random number generator
2
3  class IsingModel :
4
5      def __init__(self, beta, initialConfig):
6          self.beta = beta
7          h = len(initialConfig)
8          w = len(initialConfig[0])
9          self.initialConfiguration = zeros((h + 2, w + 2))
10         self.initialConfiguration[1:h+1,1:w+1] = initialConfig.copy()
11
12     def simulate(self, nrSteps):
13         states = deque()
14         state = self.initialConfiguration.copy()
15         states.append(state.copy())
16
17         h = len(self.initialConfiguration) - 2
18         w = len(self.initialConfiguration[0]) - 2
```

```
        for i in range(nrSteps) :
            # select a random particle
            vX = rng.integers(1, w + 1)
            vY = rng.integers(1, h + 1)

            spin = state[vX,vY]
            delta = 2*spin*(state[vX-1,vY]+state[vX+1,vY]
            +state[vX,vY-1]+state[vX,vY+1])
            if delta < 0:
                state[vX, vY] = -spin
            else:
                p = exp(-self.beta*delta)
                # sample -1 or 1 with prob p or 1-p respectively
                u = stats.bernoulli(p).rvs() * 2 - 1  # sample -1 or 1 with prob p or 1-p respect
                state[vX, vY] = u * spin
            states.append(state.copy())
        return array(states)
```

We can use this simulation to determine relevant equilibrium performance measures, such as

- At which value of β does a phase transition take place? (theoretically it can be shown that the critical value is $\beta = 1/k_B T_c = \log(1 = \sqrt{2})/2J \approx 0.44$).
- In a typical configuration, what are the fractions of $+1$ and -1 spins respectively? Related to this is the *magnetisation*, i.e. the average value of the spin.
- If a spin at any given position i is 1, what is the probability that the spin at position j is also 1?

We leave it as an exercise to the reader to find the answers to these questions. For more details we refer to [CHA87].

11.2 Wireless networks

Wireless ad-hoc networks are among the most emerging technologies. An ad-hoc network is built spontaneously as nodes (for instance laptops or mobile phones) connect. Instead of relying on some network infrastructure, the individual nodes forward packets to and from each other. The number of models to describe the dynamics of wireless communication networks has grown exponentially in recent years. However, some of the most insightful models are remarkably simple and very interesting to both analyse and simulate.

There is a striking resemblance between some of the models for wireless networks and the interacting particle systems described in the previous section. This resemblance lies in the fact that a node has two states (silent or transmitting). Moreover, whether a transition from “silent” to “transmitting” can be made, depends on the state of *all* of its neighbours, similar to the Ising model. The reason is that whenever multiple mobile devices transmit data packets simultaneously, some of these packets may get lost due to *interference*. In order to avoid packet loss, protocols have been developed such as the Carrier Sense Multiple Access (CSMA). CSMA is a probabilistic media access control (MAC) protocol in which devices (referred to as nodes) verify the absence of other transmissions before starting their own transmission on a shared medium. Before starting a transmission, each node waits for a random *backoff* time. As soon as the backoff time expires, the node checks whether all of its neighbouring nodes are silent, in order to avoid collisions. If this is the case, the node starts transmitting its data. If at least one of the neighbouring nodes was transmitting at the backoff expiration time, the node starts another random backoff time and tries again. In practice, sophisticated schemes have been developed to adapt the backoff times depending on the number of failed attempts. However, in this section we will assume a simpler model in which all backoff times are identically distributed, regardless of the number of failed attempts. We now describe the model in more detail.

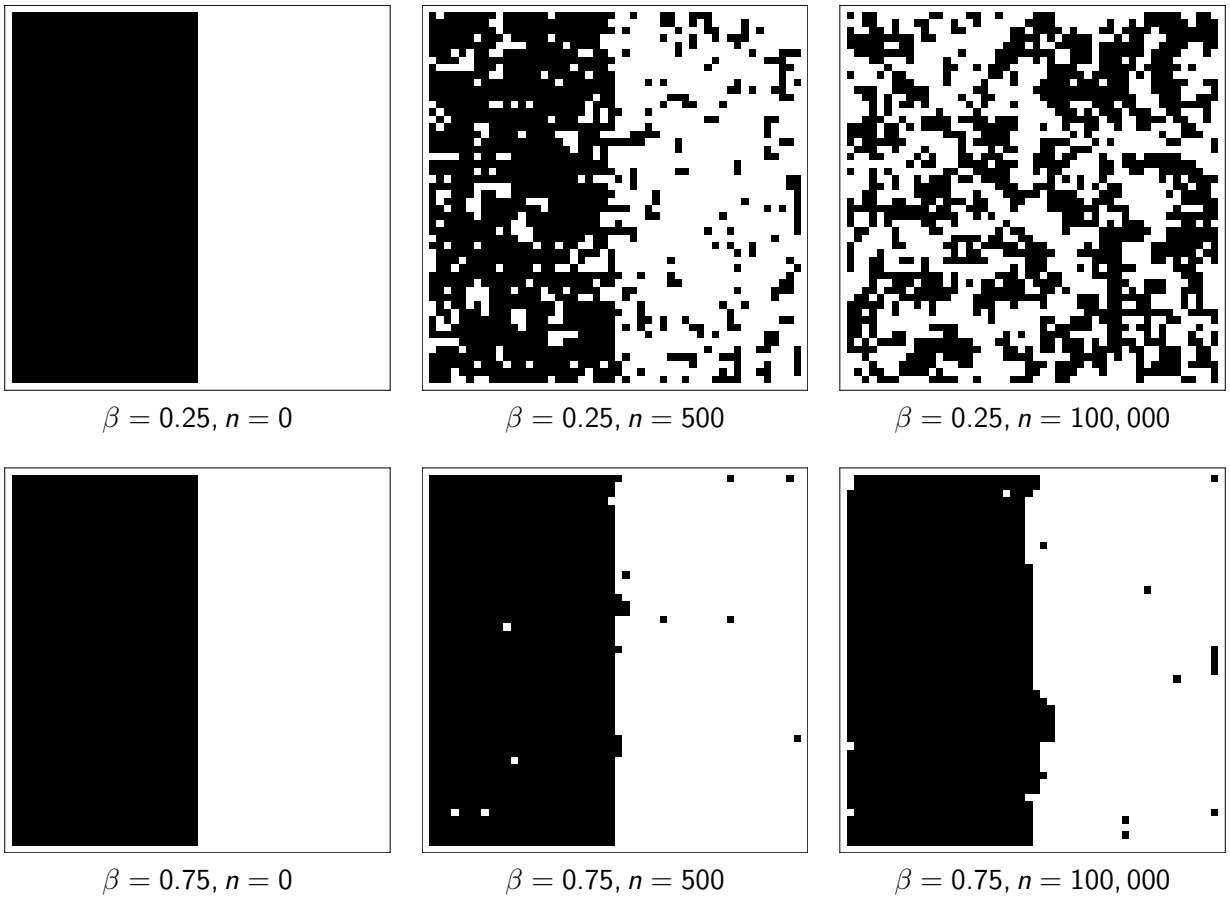


Figure 11.3: Simulated configurations for an Ising model on a 50×50 lattice. The initial configuration is $+1$ for all spin sites in the left half of the grid, and -1 for the spin sites in the right half. For $\beta > 0.44$ phase separation remains stable. But except at very high values for β (corresponding to very low temperatures), the interface fluctuates in an amoeboid fashion. At high temperatures (i.e., low values for β) the trajectory quickly moves to a highly disordered state.

Model description. We consider N nodes, scattered randomly in a square of dimension 1×1. After all, the mobility of nodes in wireless networks cannot be controlled by the network provider. Each node can forward packets to all nodes that are within a distance R. Obviously, the larger R, the better the connectivity of the network. However, a larger R means that a node should be able to transmit packets over a larger distance, and this requires more battery power (which is usually an important bottleneck in wireless settings). An example of a small network of 15 nodes is depicted in Figure 11.4.

We assume that the nodes are *saturated*, meaning that they always have packets to transmit. Obviously this is not a realistic assumption, but many mathematical models require this assumption. It can be argued that de design of a network protocol based on saturated nodes can serve as a worst-case scenario for the real-world situation, where nodes occasionally have no packets to transmit. Transmitting a packet requires a certain random time T, which we assume to be exponentially distributed with mean $\mathbb{E}[T]$. After each transmission, the node is required to wait for a backoff period B, which is also exponentially distributed, with mean $\mathbb{E}[B]$, before it can start its next transmission. As stated before, a node is only allowed to start transmitting when *none* of its neighbouring nodes are transmitting; otherwise, it has to wait for another backoff period. The main question that we want to answer using simulation is what the throughput is (number of successful transmissions per time unit), of the network and of each individual node, and how this throughput depends on N, R, $\mathbb{E}[T]$, and $\mathbb{E}[B]$.

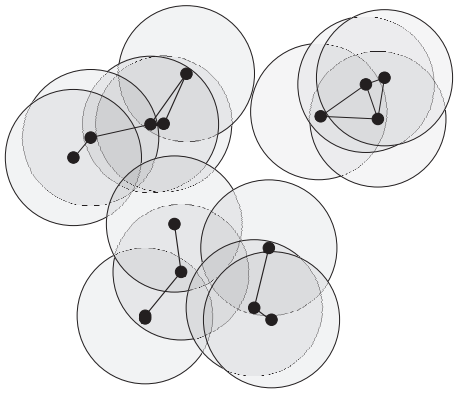


Figure 11.4: An example of a network of 15 nodes, scattered in a completely random fashion. Interference takes place within a radius R, also indicated in the figure.

Description of the simulation. Since we have a continuous-time process with multiple event types, we write a discrete-event simulation as we have done in Sections 8.3 and 8.2. Since the process is fairly simple and no queueing is involved, we conveniently skip some of the steps described in Section 8.3. The only entities that play a role are the nodes, and actually we can identify them by numbering them $1, 2, \ldots, N$. Nodes do have two states, though: either they are transmitting or not. The two events that we need to distinguish take place at the end of a backoff period and a transmission period, respectively. In order to determine the throughput at the end of the simulation, we keep track of the number of successful transmissions of each node, and divide by the total simulation time T. A compact description of the simulation is given below.

Input:

- the adjacency matrix M of dimension $N \times N$,
- the mean transmission time $\mathbb{E}[T]$,
- the mean backoff time $\mathbb{E}[B]$,
- the maximum simulation time T.

Initialisation:

- Create a vector to count the number of transmissions of each node,
- Create a vector keeping track of the state of each node (transmitting or not)
- Create the future event set,
- For each $i \in \{1, 2, \ldots, N\}$ schedule an End-Backoff event at a random exponentially distributed time with parameter $1/\mathbb{E}[B]$,
- Initialise a time variable $t = 0$.

Main simulation loop:
While $t < T$, take the next event and check whether it is an End-Backoff event or an End-Transmission event. Let i be the number of the node that is associated with this event. Update the time t to the time of the event.

End-Backoff event:

- check whether any of the neighbours of node i are currently transmitting,
- if not $\rightarrow$ change the state of node i to Transmitting, and schedule an End-Transmission event at time $t + \exp(1/\mathbb{E}[T])$ for node i,
- if yes $\rightarrow$ schedule a new End-Backoff event at time $t +$ exponential$(1/\mathbb{E}[B])$ for node i.

End-Transmission event:

- increase the counter keeping track of the number of transmissions for node i,
- change the state of node i to Not-Transmitting,
- schedule a new End-Backoff event at time $t +$ exponential$(1/\mathbb{E}[B])$ for node i.

End While.

Return the vector with transmission counts, divided by T.

The Python code to simulate this network is given in Listing 11.3. The code to generate the random graph is omitted, because it is the same as the code we used in the epidemics example, Listing 10.2. The Event and FES objects are similar to those in Listings 8.12 and 8.4, except for the fact that the Customer entity is replaced by a numeric node number.

Listing 11.3: Simulation of a wireless network.

```
class WirelessNetwork :

    def __init__(self, connectivity, backoff, transmission):
        self.connectivityMatrix = connectivity.copy()
        self.backoffDist = backoff
        self.transmissionDist =transmission

    def simulate(self, T):
        N = len(self.connectivityMatrix)
        nrTransmissions = zeros(N)
        fes = FES()
        # schedule the initial END_BACKOFF events
        for i in range(N):
            fes.add(Event(Event.END_BACKOFF, self.backoffDist.rvs(), i))
        # A vector indicating for each node whether it is currently transmitting
        isTransmitting = zeros(N)    # 0 = False
        t = 0  # current time
        while t < T:
            e = fes.next()
            t = e.time
            node = e.node
            if e.type == Event.END_BACKOFF :
                canTransmit =
                sum(self.connectivityMatrix[node] * isTransmitting) == 0
                if canTransmit :
                    isTransmitting[node] = True
                    # schedule end of transmission event
                    fes.add(Event(Event.END_TRANSMISSION,
                    t + self.transmissionDist.rvs(), node))
                else :
                    # schedule end of backoff event
                    fes.add(Event(Event.END_BACKOFF,
                    t + self.backoffDist.rvs(), node))
            else : # END_TRANSMISSION event
                nrTransmissions[node] += 1
                isTransmitting[node] = False
                fes.add(Event(Event.END_BACKOFF,
                t + self.backoffDist.rvs(), node))
        return nrTransmissions/t
```

Simulation results for an instance of $N = 50$ nodes, with radius $R = 0.2$, mean transmission time $\mathbb{E}[T] = 5$ and mean backoff time $\mathbb{E}[B] = 1$, simulated for 1000 time units, are graphically represented in Figure 11.5. The darkness of the nodes indicates the realised throughput (green = high throughput, blue = low throughput). It is noteworthy that the grid layout exhibits a nice limiting behaviour when the backoff times become small compared to the transmission times. A nice example in which this effect is extreme, is shown in Figure 11.6. We have taken a grid layout in which all nodes have four neighbours, except for the nodes at the borders of the grid. When taking extremely low backoff times, this grid turns out to have *two* limiting distributions, both shown in Figure 11.6. Both checkerboard patterns have the same probability of $1/2$ of turning up in the limit.

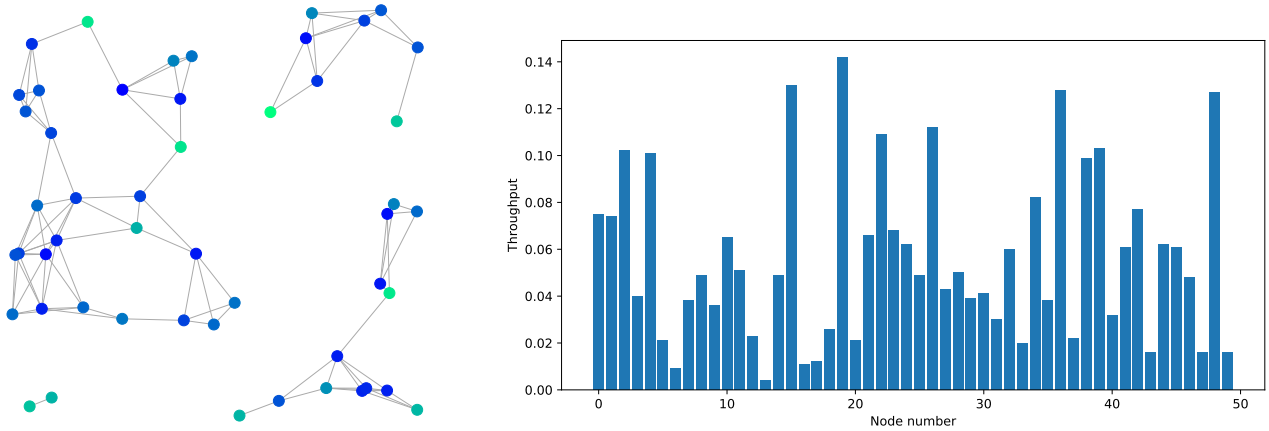


Figure 11.5: Simulated throughputs in the wireless example on a random graph with 50 nodes. On the left the graph is shown, and the node colours indicate the throughputs. The right-hand figure shows a bar chart of the simulated throughputs.

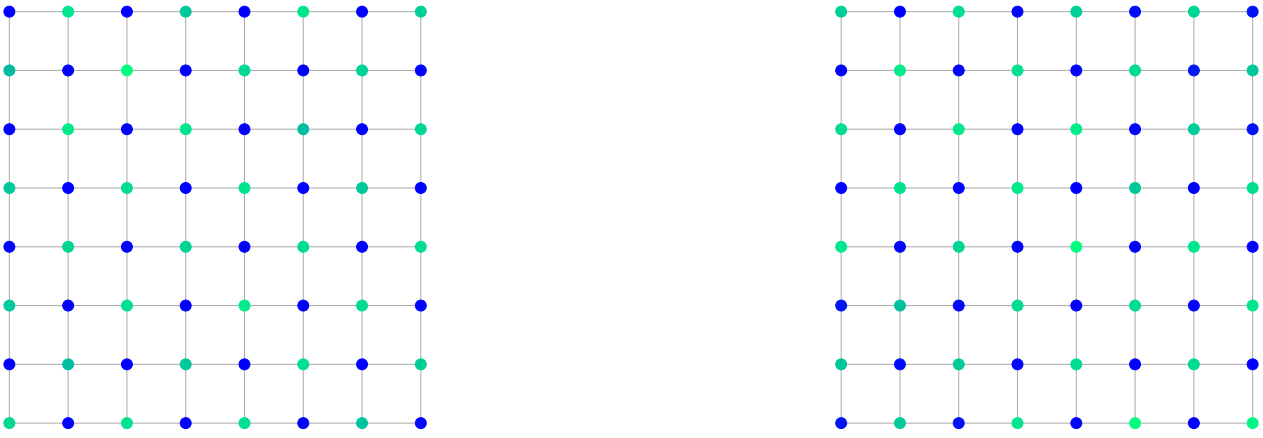


Figure 11.6: Simulated throughputs in the wireless example on a lattice with 64 nodes. This layout has two different limiting configurations as the mean backoff time approaches 0. In this example $\mathbb{E}[B] = 0.1$ and $\mathbb{E}[T] = 5$, which results in one of these two possible checkerboard structures.

12

The compound Poisson risk model

In this chapter, we will use simulation to analyze a model that is used extensively in the field of insurance mathematics. We will start with the simulation of the compound Poisson process. The compound Poisson process is the main ingredient for the compound Poisson risk process, so once we are able to simulate the compound Poisson process, we will use this for simulating the compound Poisson risk process.

Much background information on this model, which is also called the Cramér-Lundberg model, can be found in Chapter 4 of [Kaas01].

12.1 Simulating a compound Poisson process

Recall that a *Poisson process* is a counting process in which the interarrival times are independent and identically distributed according to an exponential distribution (see also Section 7.1). This basically means that our counter is set to zero at time 0 and after an exponential amount of time the counter is increased by one; then, we wait another exponential amount of time and again increase the counter by one, and so on. A *compound Poisson process* with jump size distribution G is another stochastic process, which, as the name suggests, is related to the Poisson process. If the *underlying Poisson process* is $N(t)$, the compound Poisson process can be written as

$$S(t) = \sum_{i=1}^{N(t)} X_i, \tag{12.1}$$

where the X_i are i.i.d. according to some distribution function G (which is called the jump size distribution) and independent of $N(t)$. The process can be explained as follows: consider the Poisson process. Whenever an arrival occurs, a stochastic amount or *jump* (the size of a claim, say) arrives and this is added to the compound Poisson process.

Exercise 12.1. Simulate a sample path from a compound Poisson process with rate $\lambda = 5$ and gamma distributed jump sizes with parameters $\alpha = 3.5$ and $\beta = 0.5$.

Solution:

Sample source code can be found in Listing 12.1. This code produces figures such as Figure 12.1.

Exercise 12.2. This exercise is Example 5.15 in [Ross07]. Suppose independent Exp (0.1) offers to buy a certain item arrive according to a Poisson process with intensity 1. When an offer arrives, you must either accept it or reject it and wait for the next offer to arrive. You incur costs at rate 2 per unit

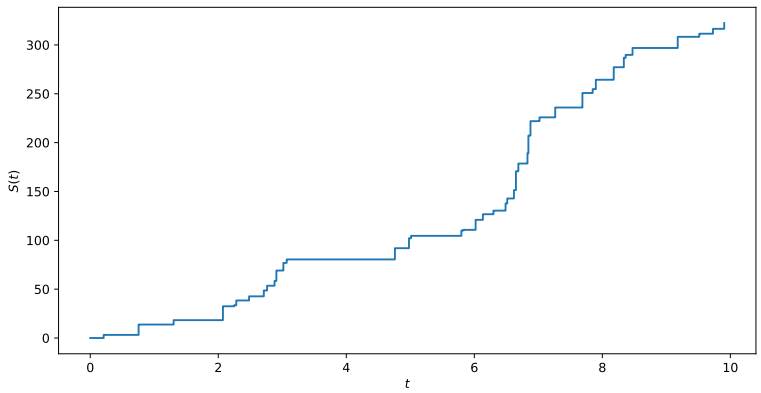


Figure 12.1: Sample path of the compound Poisson process with gamma distributed jumps.

Listing 12.1: Source code for generating sample paths from the compound Process process with exponentially distributed jumps.

```
def simulateCompoundPoissonProcess(lam, jumpDist, T):
    arrivalTimes = deque()  # the most efficient data structure
    levels = deque()  # the levels of the CPP
    currentLevel = 0
    expDist = Distribution(stats.expon(scale=1/lam))
    t = expDist.rvs()
    while t < T :
        arrivalTimes.append(t)
        currentLevel += jumpDist.rvs()
        levels.append(currentLevel)
        t += expDist.rvs()
    return arrivalTimes,levels
```

time. Your objective is to maximise the expected total return. One can think of various policies to choose which offer to accept.

(i) Accept the first offer that arrives;

(ii) Accept the first offer that is at least 9;

(iii) Accept the first offer that is at least 16;

(iv) Accept the first offer that is at least 25.

Use simulation to decide which policy is optimal.

Solution: The first policy is somewhat easier to simulate than the other three policies, since it merely involves drawing the arrival time and the size of the first offer. The other policies involve drawing a sequence of interarrival times and offers, until the first offer larger than the threshold arrives. All policies are implemented in the source code in Listing 12.2. The results, based on 10,000 independent simulation runs for each policy, can be found in Table 12.1.

The results show that the third policy is optimal, which is in accordance with Example 5.15 in [Ross07]. The example shows that the optimal policy is to accept the first offer that is larger than $10 \log 5 \approx 16.1$.

Listing 12.2: Implementation of all three policies in the search for the policy that maximises the expected total return.

```
import matplotlib.pyplot as plt
from numpy.ma.core import zeros
from scipy import stats
from dist.Distribution import Distribution

def chooseFirst(runs, arrivalRate, offerRate, costRate):
    arrival = stats.expon.rvs(scale=1/arrivalRate, size=runs)
    offer = stats.expon.rvs(scale=1/offerRate, size=runs)
    result = offer - costRate * arrival
    return result

def chooseThreshold(runs, arrivalRate, offerRate, costRate, threshold):
    arrDist = Distribution(stats.expon(scale=1/arrivalRate))
    offerDist = Distribution(stats.expon(scale=1/offerRate))
    result = zeros(runs)
    for i in range(runs):
        arrival = arrDist.rvs(1)
        offer = offerDist.rvs(1)
        while (offer < threshold):
            arrival += arrDist.rvs(1)
            offer = offerDist.rvs(1)
        result[i] = offer - costRate * arrival
    return result

runs = 10000
arrivalRate = 1
offerRate = 0.1
costRate = 2

data1 = chooseFirst(runs, arrivalRate, offerRate, costRate)
data2 = chooseThreshold(runs, arrivalRate, offerRate, costRate, 9)
data3 = chooseThreshold(runs, arrivalRate, offerRate, costRate, 16)
data4 = chooseThreshold(runs, arrivalRate, offerRate, costRate, 25)
data = [data1, data2, data3, data4]

plt.figure()
plt.boxplot(data)
plt.show()
```

Policy 1	Policy 2	Policy 3	Policy 4
7.94 ± 0.2	14.1 ± 0.2	16.3 ± 0.3	10.66 ± 0.5

Table 12.1: Approximate 95% confidence intervals based on 10,000 independent simulation runs for the three different policies.

12.2 The compound Poisson risk model

One of the main performance characteristics in insurance mathematics is the probability of ruin. In order to study this probability from a mathematical point of view, various models have been developed, of which the most well-known is probably the *compound Poisson risk model*. In this model, i.i.d. claims arrive according to a Poisson process, while if no claim arrives the capital of the insurance company grows at a constant rate: the premium per time unit.

The model can be described as

$$U(t) = u + ct - \sum_{i=1}^{N(t)} X_i. \tag{12.2}$$

Here, $U(t)$ is the capital of the insurance company at time t, c is the premium income rate, $N(t)$ is a Poisson process with rate λ. The X_i are i.i.d. claims, with common distribution function F and mean $\mu_1 := \mathbb{E}[X_1]$. Equation (12.2) can be written as

$$U(t) = u - S(t), \tag{12.3}$$

where

$$S(t) = \sum_{i=1}^{N(t)} X_i - ct. \tag{12.4}$$

The premium income rate c is often chosen such that

$$c = (1+\theta)\lambda\mu_1, \tag{12.5}$$

where θ is called the *safety loading*, with $\theta = \frac{c}{\lambda\mu_1} - 1$. This name is justified by the fact that

$$\mathbb{E}\left[\sum_{i=1}^{N(t)} X_i\right] = \lambda\mu_1 t, \tag{12.6}$$

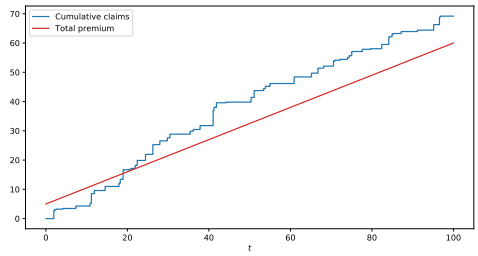
so that $\lambda\mu_1$ is the basis for the premium income rate, covering merely the expected claim per unit time. To this an extra amount $\theta\lambda\mu_1$ is added, to provide the insurance company with some extra financial room, in case a large claim arrives.

We say that the insurance company goes bankrupt (ruins) at the moment its capital $U(t)$ drops below zero. The time of ruin, denoted by $T(u)$, is then defined as

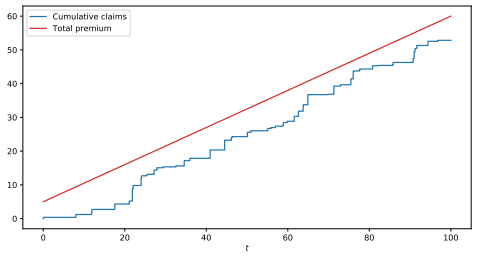
$$T(u) := \inf\{t \geq 0 : U(t) < 0\} = \inf\{t \geq 0 : S(t) > u\}. \tag{12.7}$$

If we let $\inf \emptyset = \infty$, then $T(u) = \infty$ if ruin does not occur.

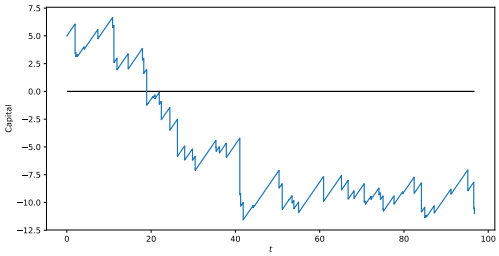
Often, the parameter u is omitted and we write $T = T(u)$. Figure 12.2 shows two sample paths for a CPP risk model with $\lambda = 0.5$, $\mu_1 = 1$, $u = 5$, and $\theta = 0.1$ (which results in $c = 0.55$). In the first path the company goes bankrupt at time 18.9, whereas in the second time the company does not go bankrupt, at least not up to time $t = 100$.



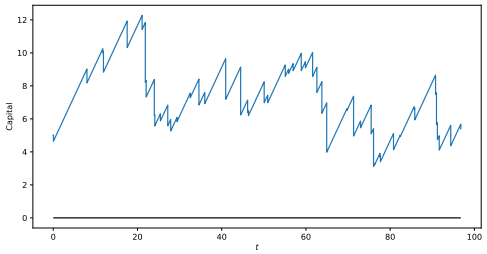
(a) ruin occurs at time $t = 18.9$



(b) no ruin up to time $t = 100$



(c) ruin occurs at time $t = 18.9$



(d) no ruin up to time $t = 100$

Figure 12.2: Two sample paths of a compound Poisson risk model with $\lambda = 0.5$, $\mu_1 = 1$, $u = 5$, and $\theta = 0.1$ (which results in $c = 0.55$). Figures (a) and (c) are two different visualisations for the same sample path. In (a) the total premium and the cumulative claim size are plotted separately. In (c) the difference of the two, i.e. the capital, is plotted. Figures (b) and (d) also belong to the same sample path.

12.2.1 Quantities of interest

The probabilities of ruin in finite or infinite time, defined as, respectively,

$$\begin{aligned} \psi(u, \tau) &:= \mathbb{P}(T \leq \tau), \\ \psi(u) &:= \mathbb{P}(T < \infty), \end{aligned}$$

are probably the most important performance characteristics in the compound Poisson risk model. Many other quantities may be of interest as well, for example the distribution of the deficit at ruin, which is the level of the process at the moment ruin occurs, given that ruin occurs, i.e.

$$\mathbb{P}(|U(T)| \leq s \mid T < \infty).$$

In order to study the probability that a company recovers from ruin, it is useful to consider quantities such as the length of ruin and the maximal severity of ruin. The first quantity is the time between the moment of ruin (T, the first time the capital of the insurance company drops below 0) and the first time after T that the capital is larger than 0 again. The second quantity denotes the minimum of the capital $U(t)$ in this time interval.

In some cases it is possible to calculate some of these quantities, while in other cases it is very hard, if possible at all. For example, when the claims follow an exponential distribution, it is well possible to derive expressions for the probability of ruin $\psi(u)$ and the distribution of the deficit at ruin. See e.g. [Kaas01].

The following result provides an upper bound for the probability of ruin. It is known as *Lundberg's upper bound*.

Proposition 12.1. *Let $M_X(t)$ be the moment generating function of the claim size distribution and let R be the positive solution of the Lundberg equation,*

$$1 + (1+\theta)\mu_1 R = M_X(R). \tag{12.8}$$

Then, for all initial capitals u, we have

$$\psi(u) \leq e^{-Ru}. \tag{12.9}$$

Python can be used to solve equations such as Equation (12.8), using the function `root`. This function finds a root of a given function, given a numeric starting value. We discussed this function in Chapter 4, see Listing 4.3.

Some risk measures admit nice formulas in the case where claim sizes follow an exponential distribution. The following result provides some of these formulas. These will later be used to compare to simulation results.

Proposition 12.2. *If the claim sizes follow an exponential distribution with mean μ_1, the following formulas hold:*

$$\psi(u) = \frac{1}{1+\theta}\exp\left(-\frac{\theta}{1+\theta}\frac{u}{\mu_1}\right),$$
$$\mathbb{P}(|U(T)| \leq s \mid T < \infty) = 1 - \exp(-s/\mu_1).$$

Note that the latter formula states that the undershoot follows an exponential distribution with mean μ_1.

12.3 Simulation

In this section, we will first focus on estimating $\psi(u,\tau)$ using simulation. Estimation of this quantity is rather straightforward. In each run, the process is simulated up to time τ and the result of the i-th simulation run Z_i is set to 1 if ruin occurs at or before time τ and to 0 if this is not the case. After n runs, our Monte Carlo estimator is then given by $\widehat{Z} = \frac{1}{n}\sum_{i=1}^{n} Z_i$.

The following algorithm can be used to obtain the result of a single simulation run.

1. Initialize the simulation: set $U = u$ and $t = 0$.
2. Draw an exponential interarrival time I with parameter λ. Draw a claim size $X \sim F$.
3. If $t + I > \tau$ return $Z_i = 0$. If not, set $U = U + cI - X$ and if $U < 0$ return $Z_i = 1$. Set $t = t + I$. Otherwise, return to step 2.

Exercise 12.3. Simulate the compound Poisson risk model and estimate the probability of ruin before time $T = 1000$, with $\lambda = 0.5$, $\theta = 0.2$, $u_0 = 5$, and claim sizes are exponentially distributed with mean 0.5.

Solution:

See the source code in Listing 12.3. An approximate 95% confidence interval, based on 10,000 independent, runs is given by 0.154 ± 0.007.

The estimation of $\psi(u,\tau)$ may be straightforward, but the estimation of $\psi(u)$ is less so. Since ruin need not occur within finite time, some of the runs may take an infinite time to finish, which makes a direct implementation impossible. However, such a simulation may be implemented approximately by choosing an appropriate stopping criterion, such as

Listing 12.3: Using Monte Carlo simulation to estimate $\psi(u, T)$ in the compound Poisson risk model.

```
def simCompoundPoissonRiskProcess(lam, claimSizeDist, u0, c, T):
    arrivalTimes = deque()   # most efficient data structure
    levels = deque()         # the levels of the CPP
    currentLevel = 0
    expDist = Distribution(stats.expon(scale=1/lam))
    t = expDist.rvs()
    ruin = False
    while t < T :
        arrivalTimes.append(t)
        currentLevel += claimSizeDist.rvs()
        ruin = ruin or (currentLevel > u0 + c*t)
        levels.append(currentLevel)
        t += expDist.rvs()
    return arrivalTimes,levels,ruin
```

1. Simulate up to some finite time T_∞ instead of $T = \infty$, and set $\psi(u) \approx \psi(u, T_\infty)$;
2. Simulate until either ruin occurs or the level u_∞ is crossed for the first time;

The idea of the first criterion, is that for large values of T_∞, the value of $\psi(u, T_\infty)$ is close to the value of $\psi(u)$. The rationale behind the second criterion is that for a high level u_∞, the probability of ruin after reaching this level becomes very small. This follows from Lundberg's exponential upper bound, $\psi(u_\infty) \leq e^{-Ru_\infty}$. So whenever the level u_∞ is reached, we may as well assume that ruin will never happen.

Remark: The first option to estimate $\psi(u)$ suggests to simulate up to some finite time T_∞. However, in Python (and many other computer algebra programs, like R, Matlab, Maple, or Mathematica) it is very time consuming to use this method, since it is unknown on beforehand exactly how many claims will arrive. Adding elements to the list `level` in Listing 12.3 is very slow. A much more efficient alternative is to specify a fixed number of claims on beforehand, say N_∞, and stop the simulation as soon as N_∞ claims have been simulated. This implementation allows the usage of very efficient functions like computing the cumulative sum and searching for an element in a list. Listing 12.4 gives a more efficient implementation. Note that this simulation continues even after ruin has occurred – and still it is faster! Listing 12.4 is more than 10 times faster than Listing 12.3.

Listing 12.4: Sample code for simulating N claims in the compound Poisson risk process. This is the most efficient implementation in many computer algebra systems.

```
def simCompoundPoissonRiskProcess2(lam, claimSizeDist, u0, c, N):
    expDist = Distribution(stats.expon(scale=1/lam))
    arrivalTimes = cumsum(expDist.rvs(N))
    levels = cumsum(claimSizeDist.rvs(N))
    premium = u0 + c * arrivalTimes
    ruin = sum(premium < levels) > 0
    return arrivalTimes,levels,ruin
```

13

Models in credit risk

This chapter is concerned with the analysis of credit risk using Monte Carlo simulation.

13.1 Credit risk measurement and the one-factor model

This section intends to be a short introduction to some of the terminology used in credit risk analysis. For more background information, we refer to [BOW03].

Banks and other financial institutions set out loans to *obligors*. These loans are subject to *credit risk*, i.e. risk induced by the fact that some of the obligors may not (fully) meet their financial obligations, for example by not repaying the loan. When this happens, we say that the obligor *defaults*, or is in default.

In order to prevent the event that defaulting obligors will let the bank go bankrupt, the bank needs some sort of insurance. Therefore, it charges a risk premium for every loan set out, and collects these in an internal bank account called an expected loss reserve, or capital cushion.

The question remains, however, how to assess credit risk in order to choose a reasonable risk premium.

The main idea is as follows: a bank assigns to each loan a number of parameters:

- The *Exposure At Default* (EAD), the amount of money subject to be lost;
- A loss fraction, called *Loss Given Default* (LGD), which describes the fraction of the EAD that will be lost when default occurs.
- A *Probability of Default* (PD).

The *loss variable* L is then defined as

$$L := \mathrm{EAD} \times \mathrm{LGD} \times \mathbf{1}_{\{D\}}, \qquad \mathbf{1}_{\{D\}} \sim \mathrm{Bin}\,(1, \mathrm{PD}) \tag{13.1}$$

Next, consider a portfolio of n obligors, each bearing its own risk. We attach a subscript i to the variables, indicating that they belong to obligor i in the portfolio. We can then define the *portfolio loss* as

$$\widetilde{L}_n := \sum_{i=1}^{n} \mathrm{EAD}_i \times \mathrm{LGD}_i \times \mathbf{1}_{\{D_i\}}, \qquad \mathbf{1}_{\{D_i\}} \sim \mathrm{Bin}\,(1, \mathrm{PD}_i)\,, \tag{13.2}$$

which is just the sum over all individual loss variables.

The distribution of $\widetilde{L}_n$ is of course of great interest, since it contains all the information about the credit risk. Several characteristics of this distribution have special names. Some of them are

- *Expected Loss* (EL), defined as $\text{EL} := \mathbb{E}\left[\widetilde{L}_n\right]$;
- *Unexpected Loss* (UL), defined as the standard deviation of the portfolio loss, $\text{UL} := \sqrt{\text{Var}\left[\widetilde{L}_n\right]}$; capturing deviations away from EL;
- *Value-at-Risk* (VaR), defined as the α quantile of the loss distribution, $\text{VaR}_\alpha := \inf\{x \geq 0 : \mathbb{P}\left(\widetilde{L}_n \leq x\right) \geq \alpha\}$. Here, α is some number in $(0, 1)$.
- *Economic Capital*, defined as $\text{EC}_\alpha = \text{VaR}_\alpha - \text{EL}$. This is the capital cushion.

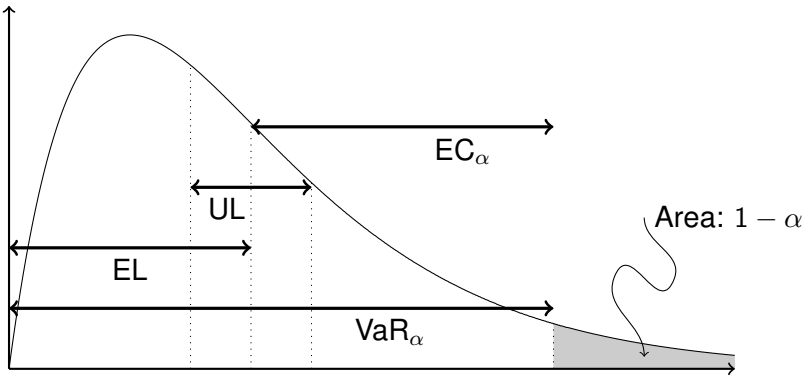


Figure 13.1: Distribution and characteristics of the portfolio loss variable $\widetilde{L}_n$.

Value-at-Risk can be used as a way to choose a suitable *capital cushion*. For example, if the capital cushion is chosen to be equal to the one-year Value-at-Risk (at a level α), the capital cushion is expected to be sufficient in $100 \times (1-\alpha)\%$ of the years. So if $\alpha = 0.998$, the portfolio loss is expected to exceed the capital cushion only twice in 1000 years.

What makes the analysis of the distribution of $\widetilde{L}_n$ interesting, is that variables $\mathbf{1}_{\{D_i\}}$ are usually correlated. This means, for example, when they are positively correlated, knowing that one of the obligors defaults, there is a higher probability of other obligors defaulting as well.

One possible model of correlated obligors in the portfolio is known as the one-factor model. In this model, the probability of default depends on the asset value at the planning horizon; in particular, we assume that there are standard normal random variables $\widetilde{r}_i$, modeling the state of the obligors at the planning horizon.[1] We will denote the critical threshold for obligor i by C_i, and we say that obligor i is in default at the planning horizon, if $\widetilde{r}_i < C_i$. It follows immediately that

$$\text{PD}_i = \mathbb{P}\left(\widetilde{r}_i < C_i\right). \tag{13.3}$$

We will now break up the variable $\widetilde{r}_i$ into two parts. One is called the *composite factor*, which, loosely speaking, captures the aspects of the real world that are of influence on the company's economic

[1] Usually, the random variable $\widetilde{r}_i$ is defined as follows. Suppose that the company's current asset value is denoted by $A_0^{(i)}$, while the value at the planning horizon T is given by $A_T^{(i)}$. We will assume that $A_T^{(i)}/A_0^{(i)}$ follows the so-called log-normal distribution. It follows that $r_i = \log(A_T^{(i)}/A_0^{(i)})$ follows a normal distribution. The values $\widetilde{r}_i$ are the standardized r_i, which means that $\widetilde{r}_i = (r_i - \mathbb{E}[r_i])/\sqrt{\text{Var}[r_i]} \sim \mathcal{N}(0,1)$.

state (for example: oil prices, the political situation, ...). The other part is the *idiosyncratic part* of the company's economic state, or, as it is often called, the *firm-specific effect*.

We will denote the composite factor by $Z \sim \mathcal{N}(0,1)$ and the idiosyncratic factor of obligor i by $Y_i \sim \mathcal{N}(0,1)$. We will assume that Z and the Y_i are independent. We choose a correlation factor $0 \leq \rho \leq 1$ and set

$$\widetilde{r}_i = \sqrt{\rho} Z + \sqrt{1-\rho} Y_i. \tag{13.4}$$

This introduces correlation into the model.

13.1.1 Simulation

If we assume that the values of EAD_i and LGD_i are given constants, a single run of a Monte Carlo simulation can be done as follows:

1. Initialize the portfolio loss, set $\widetilde{L}_n = 0$.
2. Draw a random number $Z \sim \mathcal{N}(0,1)$. This is the composite factor.
3. For each obligor $i = 1, 2, \ldots, n$:
   (a) Draw a random number $Y_i \sim \mathcal{N}(0,1)$. This is the idiosyncratic factor.
   (b) Set $\widetilde{r}_i \leftarrow \sqrt{\rho} Z + \sqrt{1-\rho} Y_i$.
   (c) If $\widetilde{r}_i < C_i$, set $\widetilde{L}_n \leftarrow \widetilde{L}_n + \mathrm{EAD}_i \times \mathrm{LGD}_i$.

Implementation

After the discussion at the start of this section, building the simulation is straightforward. An implementation of the simulation can be found in Listing 13.1. The main part of the source code is a function that performs a single simulation run and this function is executed a number of times. The loss after each run is then stored in an array and after N runs are completed, the estimated density is shown as a histogram. In this example, we choose the following values for the parameters of the model: $\rho = 0.2$, $n = 100$, $\mathrm{EAD}_i \sim \mathrm{Uniform}(0,2)$, $\mathrm{LGD}_i = 1$ and $\mathrm{PD}_i = 0.25$ for all $i = 1, 2, \ldots, n$. The number of runs is set to $N = 5000$. The result is shown in Figure 13.2

13.1.2 Estimating the value-at-risk and Tail conditional expectation

The Value-at-Risk is just the α-quantile of the loss distribution. From basic statistics, it is known that empirical quantiles can be based on an ordered random sample.

Assume that L_i, $i = 1, 2, \ldots, N$ is an independent realization from the loss distribution $\widetilde{L}_n$ (the simulation output, N is the number of runs), and denote by $L_{(i)}$ the order statistics, that is, the same sequence, but ordered in an increasing fashion, so $L_{(1)} \leq L_{(2)} \leq \ldots \leq L_{(N)}$. Now choose j such that

$$\frac{j}{N} \leq \alpha < \frac{j+1}{N}. \tag{13.5}$$

Then $L_{(j)}$ is an approximation of the $(1-\alpha)$-quantile of the loss distribution.

The expected shortfall, or tail conditional expectation, is the expected value of the portfolio loss, given that it exceeds the value-at-risk (at level α), that is,

$$\mathrm{TCE}_\alpha := \mathbb{E}\left[\widetilde{L}_n \mid \widetilde{L}_n \geq \mathrm{VaR}_\alpha\right].$$

Listing 13.1: Implementation of a simulation of the one factor model.

```
def simOneFactor(n, PD, EAD, LGD, rho):
    normDist = stats.norm(0, 1)
    z = normDist.rvs(1)  # systematic factor
    y = normDist.rvs(n)  # idiosyncratic factor
    rtilde = sqrt(rho)*z + sqrt(1-rho)*y # Asset values for obligors
    c = normDist.ppf(PD) # Critical thresholds for obligors
    default = (rtilde < c)
    losses = default * EAD * LGD
    return sum(losses)

n = 100      # The number of obligors in the portfolio.
runs = 5000 # Number of realizations.

EAD = stats.uniform.rvs(0, 2, size=n)
LGD = ones(n)
PD = 0.25 * ones(n);
rho = 0.2
losses = zeros(runs)
for i in range(runs):
    losses[i] = simOneFactor(n, PD, EAD, LGD, rho)

# Histogram total loss distribution
plt.figure()   # create a new plot window
plt.hist(losses, bins=100, density=False)
plt.show()

print(mean(losses))  # Expected loss
print(std(losses))   # Unexpected loss
```

In Listing 13.2 we show how to obtain the Value-at-Risk and the tail conditional expectation from the simulation results in Listing 13.1.

Listing 13.2: Source code used for estimating VaR_α and TCE_α.

```
alpha = 0.95
sortLosses = sort(losses)
idx = int(floor(alpha * runs))
VaR = sortLosses[idx]
print(VaR) # Value-at-risk

TCE = mean(losses[losses > VaR])
print(TCE)  # Tail conditional expectation
```

13.2 The Bernoulli mixture model

The setting of the *Bernoulli mixture model* is the same as that of the one-factor model. Again, we have n obligors, which we will call obligor $1, 2, \ldots, n$. For each obligor i, there is a random variables $\mathbf{1}_{\{D_i\}}$ indicating whether it defaults or not: $\mathbf{1}_{\{D_i\}} = 1$ means that obligor i defaults, while $\mathbf{1}_{\{D_i\}} = 0$ indicates that obligor i does not default.

We will assume that there exists some underlying random variable P, which takes values in $[0, 1]$, and that, given $P = p$, the $\mathbf{1}_{\{D_i\}}$ are independent and follow a Bernoulli distribution with parameter p:

$$\mathbf{1}_{\{D_i\}} | P \sim \text{Bin}(1, P). \tag{13.6}$$

The cdf of P is often called the *mixture distribution function*. It can be shown that for $i \neq j$ the correlation between $\mathbf{1}_{\{D_i\}}$ and $\mathbf{1}_{\{D_j\}}$ is given by

$$\text{corr}\left(\mathbf{1}_{\{D_i\}}, \mathbf{1}_{\{D_j\}}\right) = \frac{\text{Var}[P]}{\mathbb{E}[P](1 - \mathbb{E}[P])}. \tag{13.7}$$

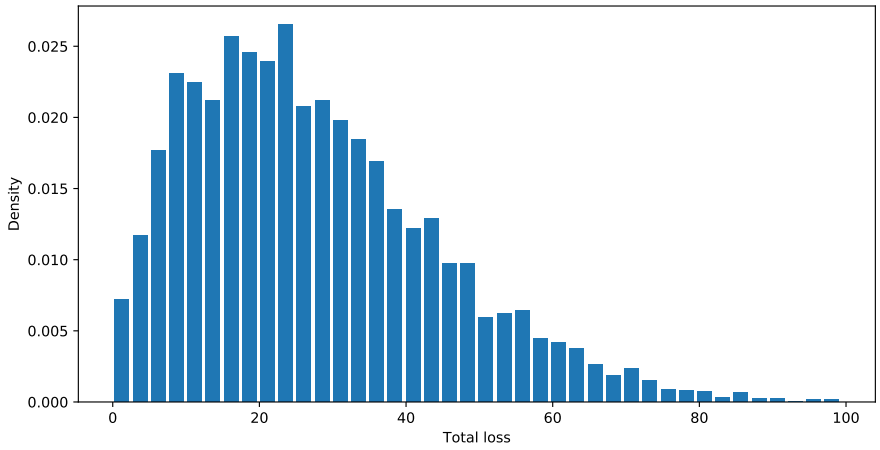


Figure 13.2: A histogram of the loss distribution in the one factor model, based on 5000 realisations.

Typical choices for the mixture distribution are the Beta distribution, or the probit-normal distribution, $\Phi^{-1}(P) \sim \mathcal{N}\left(\mu, \sigma^2\right)$, with Φ the standard normal cdf.

13.2.1 Simulation

The scheme for simulating the portfolio loss $\widetilde{L}_n$ is now straightforward:

(i) Initialize the portfolio loss, set $\widetilde{L}_n \leftarrow 0$;

(ii) Draw a single realization of P from the mixture distribution function;

(iii) For each obligor $i = 1, 2, \ldots, n$:

(a) Draw $\mathbf{1}_{\{D_i\}} \sim \text{Bin}(1, P)$;

(b) If $\mathbf{1}_{\{D_i\}} = 1$, set $\widetilde{L}_n \leftarrow \widetilde{L}_n + \text{EAD}_i \times \text{LGD}_i$.

Exercise 13.1. Write a simulation to estimate $\text{VaR}_{0.95}$ in the Bernoulli mixture model, where P is drawn from a beta distribution with parameters 1 and 3. Also, plot a histogram of the losses.

Solution: The scheme in the section "Simulation" shows how the simulation should be implemented; a sample implementation can be found in Listing 13.3. We find that $\text{VaR}_{0.95} \approx 64$. A histogram of the losses can be found in Figure 13.3.

Listing 13.3: Sample implementation of a simulation of the Bernoulli mixture model.

```
def simBernoulliMixture(n, EAD, LGD, Pdist):
    P = Pdist.rvs(1)
    binDist = stats.bernoulli(P)
    default = binDist.rvs(n)
    losses = default * EAD * LGD
    return sum(losses)

Pdist = stats.beta(1, 3)
print(Pdist.mean())   # PD = 0.25
losses = zeros(runs)
for i in range(runs):
    losses[i] = simBernoulliMixture(n, EAD, LGD, Pdist)
```

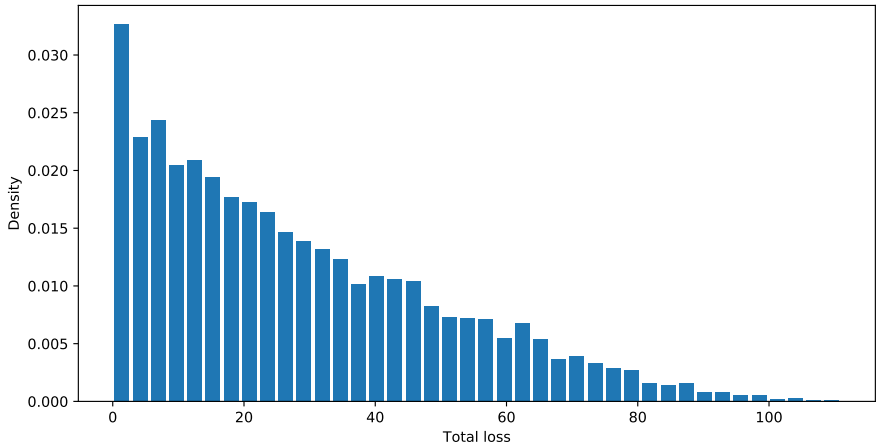


Figure 13.3: Histogram of the losses in the Bernoulli mixture model.

14

Option pricing

This chapter is devoted to option pricing using Monte Carlo simulation.

In the first section, we will give a brief overview of the underlying model (stock prices driven by a geometric Brownian motion) that we will be using and a short list of various types of options.

In the second and third section, we consider Monte Carlo simulation of the underlying process and how these techniques can be used to value different styles of options.

The fourth section describes a more advanced way of simulating Brownian motion, using wavelets.

14.1 The underlying process and options

14.1.1 Options

There exist two types of financial market instruments:

- *Stocks* – e.g. shares, bonds and commodities;
- *Derivatives* – promises of some payment or delivery in the future, that are called derivatives because their value is derived from the underlying stock (or stocks).

Stocks prices are often modelled as stochastic processes (S_t) where t is a (discrete or continuous) index indicating time. The current stock price is denoted S_0.

Options are a specific type of derivative, that give the holder the right (but not the obligation) to buy or sell a certain amount of the underlying stock on (or sometimes before) a specified date (the *expiration date* or *maturity*) for a specified price (the strike price. We will denote the maturity by T and the strike price by K.

Options that give the holder the right to buy are called *call options*, while options that give the holder the right to sell are called *put options*. When an option is used to buy or sell stock, we say that it is exercised.

Options come in many flavours, of which we will mention the two that are most well-known.

(i) *European options.* These options give the holder the right to buy (call option) or sell (put option)

a number of units of the underlying stock on time T for strike price K. The pay-off of a European call option is $(S_T - K)^+$ and that of a put option is $(K - S_T)^+$, where $(x)^+ = \max(x, 0)$.

(ii) *American options.* These are like European options, but they can be exercised at or before time T.

14.1.2 Modelling the stock price

A model that is often used for financial markets is the Black-Scholes model that is named after Fisher Black and Myron Scholes, who described the model in their 1973 paper "The pricing of options and corporate liabilities" [Bla73]. Black and Scholes model the stock price as a geometric Brownian motion. That is, given the current stock price S_0, the stock price process $(S_t)_{t\geq 0}$ is such that

$$\log\left(\frac{S_t}{S_0}\right) \tag{14.1}$$

is a Brownian motion.

14.1.3 Valuation of options

An important problem in mathematical finance is the valuation – or pricing – of derivatives. At the moment that a derivative is taken out, it is not known what its value will be in the future, as it depends on the future value of the underlying stock.

14.2 Simulating geometric Brownian motion

In Section 7.2 we have shown how to simulate a Brownian motion. However, a Brownian motion does not seem to be adequate as a model for stock markets, because, for example, it would allow for negative stock prices. The Black-Scholes model uses a *geometric Brownian motion* instead. If we denote, as before, by $(B_t)_{t\geq 0}$ a standard Brownian motion, then the process

$$S_t = S_0 \exp(\mu t + \sigma B_t) \tag{14.2}$$

is a geometric Brownian motion with drift μ and volatility σ. $(S_t)_{t\geq 0}$ will be the model of our stock prices.

As before, the simple formula (14.2) allows us to simulate a geometric Brownian motion, as soon as we are able to simulate the standard Brownian motion.

14.2.1 The martingale measure

Pricing options in the GBM model can be done using the risk-neutral measure. Under this measure, the stock price follows a geometric Brownian motion with drift $r - \sigma^2/2$ and volatility σ. Here, r is the continuously compounded interest rate.

A general option has a pay-off function that depends on the sample path of the underlying Brownian motion, say

$$\text{pay-off} = F((S_t)_{0\leq t\leq T}) \tag{14.3}$$

for some function G. The price of this option then will be

$$C = \mathrm{e}^{-rT}\mathbb{E}\left[F((S_0 \exp((r - \sigma^2/2)t + \sigma B_t)_{0\leq t\leq T})\right], \tag{14.4}$$

with $(B_t)_{t\geq 0}$ a Brownian motion and r is the continuously compounded interest rate. For details, see e.g. [Etheridge02] or the lecture notes by Harry van Zanten [Zanten10].

14.3 Option pricing in the GBM model

14.3.1 European options

Pricing European options using Monte Carlo simulation is relatively easy, as the price of the option only depends on the stock price path $(S_t)_{0\leq t\leq T}$ through the value S_T: the stock price at maturity. For now, we will focus on the European call options, but simulating put options works just the same.

In the case of European call options, the function F from the previous sections will be

$$F((S_t)_{0\leq t\leq T}) = (S_T - K)^+, \tag{14.5}$$

with K the strike price of the option. In other words: the exact dynamics of the underlying process are not important, we only care about the value

$$S_T = S_0 \exp((r - \sigma^2/2)T + \sigma B_T), \tag{14.6}$$

which is easily generated, once we are able to generate the value of the standard Brownian motion at time T, which is just an $\mathcal{N}(0, T)$-distributed random variable! Sample source code for a Monte Carlo algorithm for pricing European call options can be found in Listing 14.1.

Listing 14.1: A Monte Carlo algorithm for pricing European call options in the GBM model.

```
def stockPriceEuropeanCall(r, sigma, T, S0):
    Z = stats.norm.rvs(0, sqrt(T), size=1)
    return S0 * exp((r - sigma * sigma/2) * T + sigma*Z)

def payoffEuropeanCall(s, K):
    return max(s - K, 0)

T = 5           # Time horizon
r = 0.03        # Interest rate
sigma = 0.05    # Volatility
runs = 10000    # Number of runs
S0 = 1          # Initial stock level
K = 1.01        # Strike price

results = zeros(runs)
for i in range(runs):
    s = stockPriceEuropeanCall(r, sigma, T, S0);
    p = payoffEuropeanCall(s, K)
    results[i] = exp(-r * T) * p

# Mean and variance
print(mean(results))
print(var(results))
```

Exercise 14.1. Adapt the source code in Listing 14.1 so that the algorithm prices European put options instead of European call options.

Solution: To price put options instead, change `return max(s-K, 0)` into `return max(K-s,0)`.

Exercise 14.2. A binary put option is an option that pays out one unit of cash if the stock price at maturity is below the strike price. Adapt the source code in Listing 14.1 so that it prices such a binary put option.

Solution: To price the binary put option, it suffices to replace the `payoff(s, K)`-function, see Listing 14.2.

Listing 14.2: Pay-off function for a binary put option.

```
def payoffBinaryPut(s, K):
    p = 1
    if s > K:
        p = 0
    return p
```

14.3.2 General options

The general version of Equation (14.3) shows that the payoff of an option may depend on the whole sample path, rather than only on the value of the stock at maturity, as is the case with European options. Pricing a general option therefore needs simulation of the whole sample path of a geometric Brownian motion rather than only its value at maturity.

We can adapt the "discretisation" strategy for simulating sample paths of the standard Brownian motion to simulate sample paths of the geometric Brownian motion. As an example, we will consider Asian call options. The value of an Asian call option is based on the average price of the underlying over the time until maturity. Its pay-off may be given as the maximum of the average value over the time until maturity minus the strike, and zero. A sample implementation can be found in Listing 14.3, where we have used the cumulative product `cumprod` function to efficiently compute the sample path.

Listing 14.3: A Monte Carlo algorithm for pricing Asian call options in the GBM model.

```
def stockpriceAsianCall(r, sigma, T, M, S0):
    dt = T/M # Time step
    Z = stats.norm.rvs(0, sqrt(dt), size=M)
    terms = S0 * append([1], exp((r - sigma**2/2) * dt + sigma * Z))
    path = cumprod(terms)
    return path

def payoffAianCall(s, K):
    return max(mean(s) - K, 0)
```

Exercise 14.3. Adapt the source code in Listing 14.3 so that it can be used to price lookback put options that expire in 5 months and look back two months.

A lookback option gives the holder the right to buy/sell the underlying stock at its lowest/highest price over some preceding period.

Solution: To price the binary put option, we need to extend the `payoff(s, K)`-function. It needs to find the minimum value in the second half of the list of sample points `s`, starting from the index corresponding to the moment that the lookback period starts (in this case, after three months), for which we will use

$$\frac{3}{5} \times \#\text{sample points} = \frac{t}{T} \times \#\text{sample points}, \tag{14.7}$$

rounded up. See Listing 14.4.

Listing 14.4: Pay-off function for a lookback put option.

```
1  def payoffLookbackPut(s, K, T, t):
2      index = ceil((t / T) * len(s))
3      lowest = min(s[index:len(s)])
4      return s[len(s) - 1] - lowest
```

Part IV

Advanced Topics

15

Diffusion limits

This chapter is still under construction. It might contain mistakes. If you find one, please let us know!

In this chapter we will not introduce new models to simulate, but instead we show a connection between some of the models discussed in the previous chapters. The connection occurs when systems operate under extreme conditions, such as very heavy load, or extremely high arrival rates. It will be shown that under these conditions, the limiting behaviour of discrete-event systems, such as random walks and queueing systems, becomes a so-called *diffusion process*, such as a Brownian motion or an Ornstein-Uhlenbeck process. A seminal book on diffusion limits is written by Ward Whitt [WW02], giving much more detailed information. Many examples in the first section are inspired by the first chapter of this book.

15.1 Random walk

In this section we revisit the simple random walk, discussed in Section 6.1. Recall that the simple random walk is a stochastic process $\{S_n, n = 0, 1, 2, \dots\}$ defined as follows,

$$S_0 = 0, \quad S_n = \sum_{i=1}^{n} X_i, \qquad n = 1, 2, \dots,$$

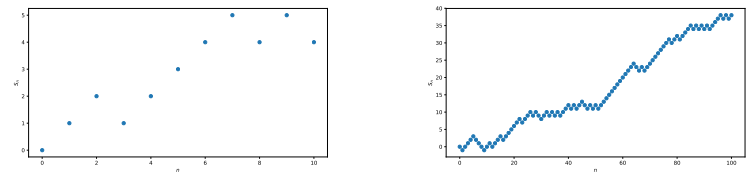
where $X_1, X_2, \dots$ are independent identically distributed random variables with $\mathbb{P}(X_i = 1) = p$ and $\mathbb{P}(X_i = 1) = 1 - p$. In Figure 15.1 we have plotted four random sample paths for a random walk with $p = 3/4$ and $n = 10, 100, 1000, 10000$.

When studying Figure 15.1, one immediately notices that the "randomness seems to vanish" for increasing values of n. Obviously, this is not true at all, since the standard deviation of S_{10000} is 10 times higher than the standard deviation of S_{100}. Nevertheless, it is clear that the plots seem to become more independent of n as n increases, converging to a straight line with slope 0.5 (which is the mean of the increments X_i). The Python code to plot the sample paths is simple. We simply plot the simulated values of S_k for $k = 0, 1, \dots, n$ and let the plotter decide how to scale the plot. For values above $n = 10,000$, the plots will hardly change at all if one only considers the shape of the sample path, ignoring the scales of the x-axis and y-axis. The fact that the rescaled versions of the plot look like the original plot indicates *self-similarity*.

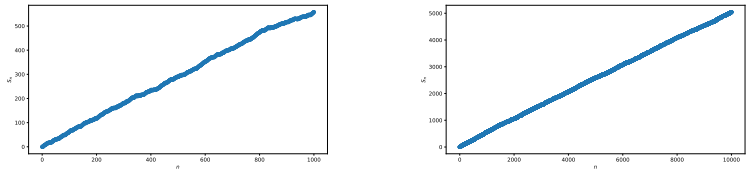
The Central Limit Theorem (see Chapter 2) tells us how plots should be scaled for increasing values of n. For large values of n, the sum S_n may be approximated by a normal distribution with mean and standard deviation

$$\mathbb{E}[S_n] = n\mu_X, \qquad \mathrm{sd}[S_n] = \sqrt{n}\,\sigma_X,$$

where μ_X and σ_X are the mean and standard deviation of the increments X_i respectively. This tells us exactly how the x and y-axes of the plots scale, namely with a factor n and $\sqrt{n}$ respectively. In



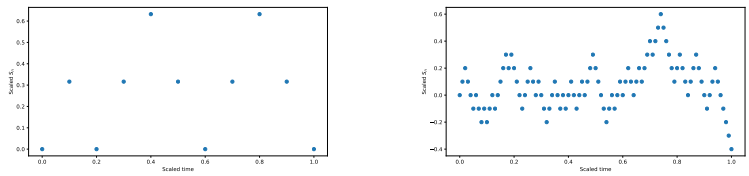
(a) $n = 10$ (b) $n = 100$



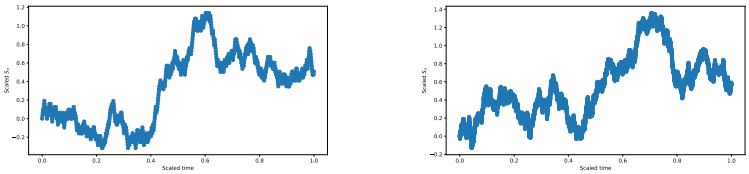
(c) $n = 1000$ (d) $n = 10000$

Figure 15.1: Four sample paths of a random walk with $p = 3/4$, for different values of n.

order to obtain standardised plots, we should divide the x-values by n and the y-values by $\sqrt{n}$, which is referred to as scaling of *time* and *space*. In Figure 15.2 we have plotted four random sample paths for a random walk with $p = 1/2$ and $n = 10, 100, 1000, 10000$. For $p = 1/2$, the random walks have no drift anymore. To the human eye this makes them appear "more random" for large values of n than the random walk with drift, shown in Figure 15.1. However, these plots exhibit the same kind of regularity from a statistical point of view.



(a) $n = 10$ (b) $n = 100$



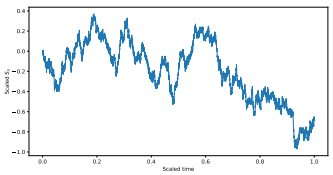
(c) $n = 1000$ (d) $n = 10000$

Figure 15.2: Four sample paths of a random walk with $p = 1/2$, for different values of n. The x and y axes of the plots are scaled with a factor n and $\sqrt{n}$ respectively.

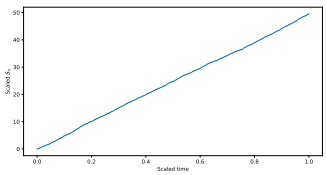
Diffusion limits. Since the scaled random walks are practically identical for large values of n, it should not come as a surprise that it can be shown rigorously that for $n \to \infty$ the scaled random walks converge to a fixed stochastic process. This stochastic-process limit is referred to as a *diffusion limit*. It is beyond the scope of this stochastic simulation course to provide a rigorous derivation of the limiting process. Instead we argue intuitively that for $n \to \infty$ the interval $[0, 1]$ will be divided into infinitely small subintervals of length $1/n$, with corresponding increments X_i. The huge number of increments at infinitely small subintervals behave, in the limit, as normally distributed random variables. The resulting diffusion limit can be shown to be a *Brownian motion*, discussed in Section 7.2. Let $\widetilde{S}_n := S_n/\sqrt{n}$ be the (space) scaled random walk. Then

$$\mathbb{E}[\widetilde{S}_n] = \sqrt{n}\mu_X, \qquad \mathrm{sd}[\widetilde{S}_n] = \sigma_X.$$

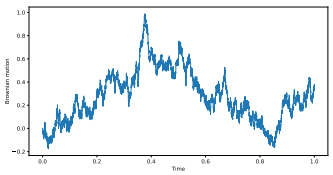
If we also scale time, by a factor n, and let $n \to \infty$, we obtain a Brownian motion on the interval $[0, 1]$ with drift μ_X and volatility σ_X. To illustrate this graphically, we have plotted the same random walks with $n = 10,000$ as in Figures 15.1(d) and 15.2(d), with time and space scaled and using lines instead of points, and sample paths of a Brownian motion with corresponding drifts and volatilities, in Figure 15.3. Although the paths themselves are obviously different, it does appear that they are from the same process and, indeed, act on the same space and time scales. The Python code to generate these plots is given in Listing 15.1. We use the functions to simulate random walks and Brownian motions from Chapters 6 and 7.



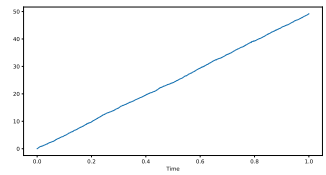
(a) Random walk $p = 1/2$



(b) Random walk $p = 3/4$



(c) Brownian motion $\mu = 0$, $\sigma = 1$



(d) Brownian motion $\mu = 50$, $\sigma = \sqrt{3/4}$

Figure 15.3: A simple random walk with $10,000$ increments and corresponding Brownian motions.

Listing 15.1: Python code to simulate the random walks and corresponding limiting Brownian motions of Figure 15.3.

```
def plotScaledRandomWalk(p, n):
    rw = simRandomWalk2(p, n)/sqrt(n)
    plt.figure()
    plt.plot(arange(n+1)/n, rw)

def plotBM(mu, sigma, T, M):
    dt = T/M
    bm = simulateBrownianMotion(mu, sigma, T, M)
    plt.figure()
    plt.plot(arange(0, T+dt, dt), bm)

n = 10000
plotScaledRandomWalk(0.5, n)
plotScaledRandomWalk(0.75, n)
plotBM(0.0, 1, 1, n)
plotBM(0.5*sqrt(n), sqrt(0.75), 1, n)
plt.show()
```

15.2 The $G/G/1$ queue in heavy traffic

Recursion:

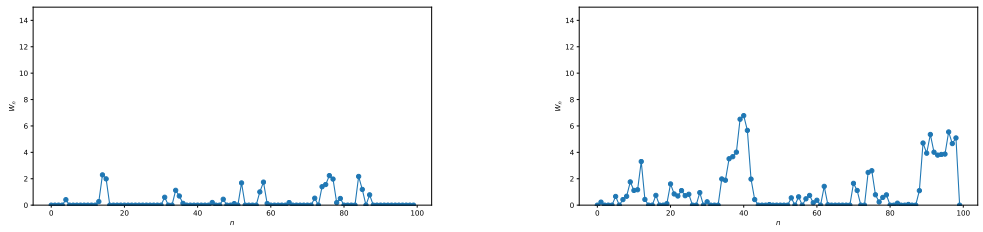
$$W_{n+1} = \max(W_n + B_n - A_n, 0).$$

Parameters:

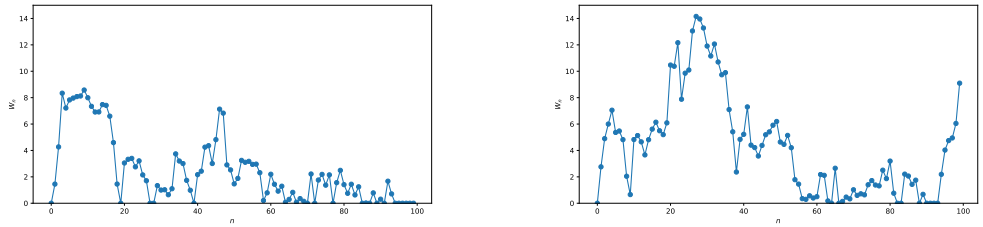
$$\alpha = \mathbb{E}[A - B] = \frac{1}{\lambda} - \frac{1}{\mu}, \qquad \beta^2 = \mathrm{Var}\,[A - B] = \frac{1}{\lambda^2} + \frac{1}{\mu^2},$$

which results in a scaling parameter of

$$\frac{\beta^2}{2\alpha}.$$

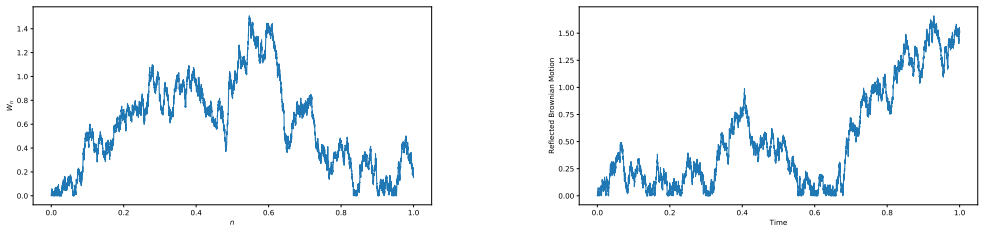


(a) $\rho = 0.4$ (b) $\rho = 0.6$



(c) $\rho = 0.8$ (d) $\rho = 0.95$

Figure 15.4: Four simulations of 100 waiting times in an $M/M/1$ queue, for different values of ρ.



(a) $G/G/1$ queue with $\rho = 0.99$ (b) Reflected Brownian motion

Figure 15.5: Simulated scaled waiting times in an $M/M/1$ queue and a corresponding reflected Brownian motion.

15.3 The $M/M/s$ queue in the Halfin-Whitt regime

$$\lambda = s - \gamma\sqrt{s}, \qquad \text{for } \gamma > 0.$$

Without loss of generality, we assume that $\mu = 1$ in this section. By increasing the number of servers and, accordingly, the arrival intensity, the load of the system will tend to one:

$$\rho = \frac{\lambda}{s\mu} = 1 - \frac{\gamma}{\sqrt{s}} \uparrow 1, \qquad (\lambda, s \to \infty).$$

We want to know what the limiting process is as we increase s to infinity. Consider the diffusion process of the scaled queue length,

$$D(t) = \frac{X(t) - s}{\sqrt{s}}.$$

In order to determine the drift of this diffusion process, we study the transition diagram of the original queue-length process again, shown in Figure 15.6. From this diagram it is apparent that the drift from state k to state $k+1$ is always equal to the arrival intensity λ. The drift from k to $k-1$ is different for $k < s$ than for $k \geq s$. In the latter case, all s servers are occupied, resulting in a negative drift of $-s$ (since we assume $\mu = 1$). The drift for $k < s$ is equal to the number of occupied servers times $-\mu$, which is equal to $-k$. If we now consider the infinitesimal drift $d(x)$ of the diffusion process at a

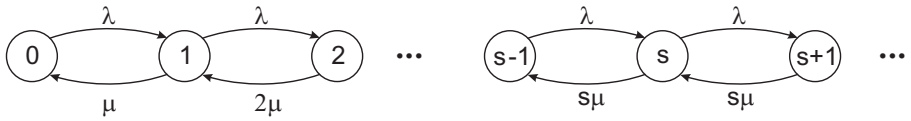


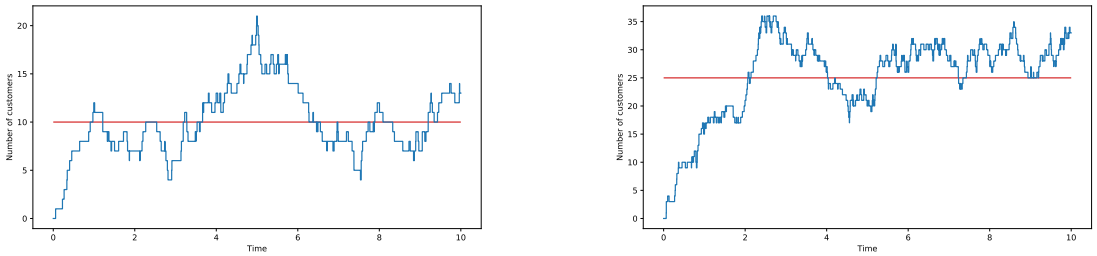
Figure 15.6: Transition diagram for the $M/M/s$ queue.

certain level x, and we distinguish the cases $x < 0$ and $x > 0$, we have:

$$d(x) = \begin{cases} \frac{\lambda - s}{\sqrt{s}} = -\gamma & \text{for } x \geq 0, \\ \frac{\lambda - (x\sqrt{s}+s)}{\sqrt{s}} = -\gamma - x & \text{for } x < 0. \end{cases}$$

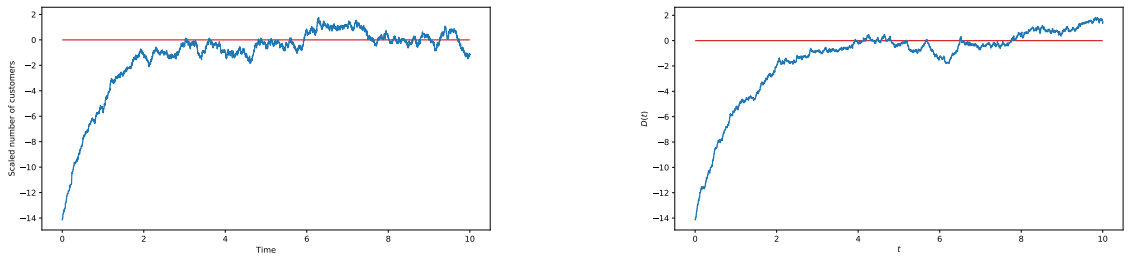
Note that the drift does *not* depend on x for $x > 0$. It can be shown that for $x > 0$ the diffusion process $D(t)$ is a Brownian motion with drift $-\gamma$. For $x < 0$ the drift increases as x decreases further. In this case it can be shown that $D(t)$ is an Ornstein-Uhlenbeck process with drift $-(\gamma + x)$. Combined, this results in the following stochastic differential equation for $D(t)$,

$$\mathrm{d}D(t) = \big(-\gamma - \min(D(t), 0)\big)\mathrm{d}t + \mathrm{d}B(t).$$



(a) $s = 10$ (b) $s = 25$

Figure 15.7: Simulated queue-length sample paths of an $M/M/s$ queue with varying numbers of servers, and arrival intensities $\lambda = s - 0.2\sqrt{s}$.



(a) The $M/M/s$ queue in the Halfin-Whitt regime (b) The diffusion limit $D(t)$

Figure 15.8: Simulated scaled queue-length sample path of an $M/M/s$ queue in the Halfin-Whitt regime for $s = 200$, and the diffusion limit.

16

Importance Sampling

This chapter is still under construction. It might contain mistakes. If you find one, please let us know!

16.1 Rare Events: Efficiency Issues

Rare events are, as the name implies, events that are rare, in the sense that they have small probability, say of order 10^{-3}, or 10^{-9}. Probabilities as small as these may not look very interesting at first sight. Indeed, if events have such small probabilities, they do not happen very often, so why would one be bothered? On the other hand, rare events occur quite naturally in many settings. One such setting we have already encountered in Chapter 12, where we estimated the probability of ruin in infinite time, $\psi(u)$. Other examples can be found in such areas as network analysis (computer networks are often modeled as networks of queues with finite buffers. When a packet arrives at a queue of which the buffer is full, the packet is lost. The probability of interest may be that of packet loss, which is typically of order 10^{-9} and security (naturally, one would want the probability of melt-down of a nuclear plant to be low). These events have two things in common: their probabilities are extremely small, and if they occur, their consequences are catastrophic.

From a theoretical point of view, estimation of the probability of rare events is interesting as well. It turns out that (Crude) Monte Carlo (CMC) simulation (in which the estimator equals one if the rare event occurred and zero otherwise) is highly inefficient. To see why, consider the networking example from above, in which the probability of packet loss is of order 10^{-9}. This means that in a CMC simulation, approximately one in every one billion packets will be lost. In order to obtain a good estimate of the probability, multiple billions of runs will have to be executed. Typically, in each run, many random variables have to be drawn. This will of course be very time consuming (if possible at all).

Another aspect is the accuracy of the estimate. Suppose that we want to estimate the probability $z = \mathbb{P}(A)$ of a rare event A. The realizations Z_i of a Crude Monte Carlo simulation take on the value 1 if the event occurred in that run, and zero otherwise. Therefore, the random variables Z_i follow a Bernoulli distribution with probability parameter z. The half-width of the confidence intervals is proportional to $\sqrt{\mathrm{Var}[Z]}$ (cf. Equation (2.14)). Since Z follows a Bernoulli distribution, we have $\mathrm{Var}[Z] = z(1-z)$ and it is easily seen that as z tends to zero, $\mathrm{Var}[Z]$, and therefore the absolute error, tends to zero as well.

However, the absolute error is not that interesting to use as a performance measure. To see why, suppose that we have found an estimate for z of order 10^{-5} and a confidence interval of half-width 10^{-4}. Although this confidence interval does look narrow (absolute error), it does not help to tell whether z is of order 10^{-4} or 10^{-5}. The relevant performance measure is the relative error $\sqrt{\mathrm{Var}[Z]}/z$.

To analyze the behavior of this quantity, note that

$$\frac{\sqrt{\text{Var}\,[Z]}}{z} = \frac{\sqrt{z(1-z)}}{z} = \frac{\sqrt{1-z}}{\sqrt{z}} \sim \frac{1}{\sqrt{z}} \to \infty, \tag{16.1}$$

as $z \downarrow 0$. This means that when the probability of interest, z, tends to zero, the relative error of the estimate will tend to infinity!

The precision of an estimate can be measured in terms of half-width of the 95% confidence interval, obtained from N independent runs. It is then given by

$$\frac{1.96\frac{\sqrt{\text{Var}[Z]}}{\sqrt{N}}}{z}. \tag{16.2}$$

Note that it is proportional to the relative error. If we want to find the number N of runs that is needed to obtain a given precision (say 10%), we have to set the expression in (16.2) equal to 0.1 and solve for N. This yields

$$N = \frac{100 \times 1.96^2 z(1-z)}{z^2}, \tag{16.3}$$

which tends to infinity whenever $z \downarrow 0$. This justifies the remark about inefficiency earlier in this Section.

In the following, let $A(x)$ be a family of events indexed by $x \in (0, \infty)$, such that $z(x) := \mathbb{P}(A(x)) \to 0$ as $x \to \infty$. For example, in Chapter 12, the events $A(x)$ would be

$$A(x) = \{\text{ruin occurs, given that the initial capital is } x\} \tag{16.4}$$

and $z(x) = \psi(x) = \mathbb{P}(A(x))$ and we know that $z(x) \to 0$ if $x \to \infty$.

In this setting, let $Z(x)$ be an unbiased estimator for $z(x)$. The discussion about efficiency above motivates the following definitions:

Definition 16.1. The estimator $Z(x)$ is said to have **bounded relative error** as $x \to \infty$, if and only if

$$\limsup_{x\to\infty} \frac{\text{Var}\,[Z(x)]}{(z(x))^2} < \infty. \tag{16.5}$$

Definition 16.2. The estimator $Z(x)$ is said to have **logarithmic efficiency**, if and only if for some $\varepsilon > 0$

$$\limsup_{x\to\infty} \frac{\text{Var}\,[Z(x)]}{(z(x))^{2-\varepsilon}} = 0 \tag{16.6}$$

or equivalently

$$\liminf_{x\to\infty} \frac{|\log \text{Var}\,[Z(x)]|}{|\log (z(x))^2|} \geq 1. \tag{16.7}$$

It is not hard to show that bounded relative error implies logarithmic efficiency.

16.2 Importance Sampling

Let X be a random variable with density f and suppose that we are interested in estimating

$$z := \mathbb{E}[X] = \int x f(x) \mathrm{d}x \tag{16.8}$$

(if X is a discrete random variable with probability mass function, the integral can be replaced by a sum).

If Y is another random variable with density g, which takes values in the same area as X (so $f(x) = 0$ implies $g(x) = 0$), we can write z as

$$z = \int \frac{yf(y)}{g(y)} g(y) \mathrm{d}y = \mathbb{E}_g \left[\frac{Yf(Y)}{g(Y)} \right], \tag{16.9}$$

where we use the subscript g in $\mathbb{E}_g[\cdot]$ to indicate that Y has density g, and the expectation is taken with respect to density g. The quotient $f(Y)/g(Y)$ is often written as $L(Y)$; the function L is called the likelihood ratio.

In this discussion, the random variables X can be replaced by functions of several random variables, say $X = h(X_1, X_2, \dots, X_m)$. In this case, the density f should of course be replaced by the joint density of $X_1, X_2, \dots, X_m$,

$$f(x) = f(x_1, x_2, \dots, x_m). \tag{16.10}$$

Bottom line of this discussion is that we now have two ways of estimating z:

(i) Draw realizations $X_1, X_2, \dots, X_n$ from density f and use as an estimator $(X_1 + X_2 + \dots + X_n)/n$.

(ii) Draw realizations $Y_1, Y_2, \dots, Y_n$ from density g and use as an estimator $(Y_1' + Y_2' + \dots + Y_n')/n$, where $Y_i' = Y_i f(Y_i)/g(Y_i)$.

The second approach is known as **importance sampling**. In some cases, direct simulation of X to obtain an estimate of z may not be convenient, because it is difficult to simulate X, or the variance of X is large, or both. In this case, importance sampling may be a good alternative.

This leads us to the question which density g to choose in order to obtain estimates with a smaller variance. A good choice of g could decrease the estimator variance by orders of magnitude. However, a bad choice of g potentially blows up estimator variance by orders of magnitude!

The art of importance sampling is choosing a good density function g from which to draw random variables for the simulation. Recall that if we want to estimate $\mathbb{E}[h(X)]$ the importance sampling estimator is given by

$$Z_g = \frac{1}{n} \sum_{i=1}^{n} \frac{h(Y_i) f(Y_i)}{g(Y_i)}, \qquad \text{with } Y_i \sim g. \tag{16.11}$$

It can be shown, using Jensen's inequality, that the variance of Z is minimized when the fraction $h(Y_i)f(Y_i)/g(Y_i)$ is constant. In fact, if the function g can be chosen such that $h(y)f(y) = \alpha g(y)$ for all x and for some constant α, the variance of the estimator is zero. In this case, we would need only a single realization and set $Z_g = h(Y_1)f(Y_1)/g(Y_1)$. However, in this case we have would have $Z_g = \alpha$, and since Z_g is an unbiased estimator, we would have $\mathbb{E}[h(X)] = \alpha$. This shows that if we choose g in this way, the value of $\mathbb{E}[h(X)]$ would have to be known in the first place. But estimating this value was the reason for using Monte Carlo simulation in the first place!

However, the discussion above shows that a good estimator can be obtained by choosing g in such a way that $h(x)f(x)/g(x)$ is approximately constant, which means that the shape of $g(x)$ is close to the shape of $h(x)f(x)$.

Example 16.1. This example is taken from [And99]. Suppose that the function h is given by $h(x) = x(1-x)^3$, with $0 \le x \le 1$. A plot of this function can be found in Figure 16.1. Suppose that we want to use numerical integration using Monte Carlo simulation to estimate $\int_{x=0}^{1} h(x)\mathrm{d}x$.

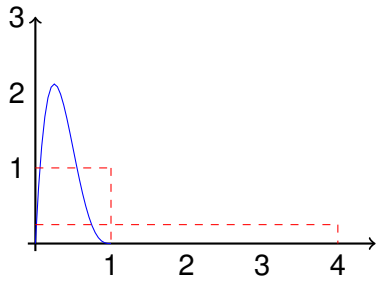


Figure 16.1: Plot of the function $g(x) = x(1-x)^3$ and two uniform pdfs.

Straightforward Monte Carlo simulation would use Uniform $(0,1)$ random variables $U_1, U_2, \ldots, U_n$ and use as an estimator $\frac{1}{n}\sum_{i=1}^{n} h(U_i)$.

The plot, however, suggests another approach. One could use Uniform $(0,4)$ random variables $V_1, V_2, \ldots, V_n$. Note that

$$\int_{x=0}^{1} h(x)\mathrm{d}x = 5\mathbb{E}\left[h(V_1)\right], \tag{16.12}$$

so we could use as an estimator $\frac{5}{n}\sum_{i=1}^{n} h(V_i)$. Obviously, the latter option makes no sense, since about 75% of the random variables V yields no information whatsoever about the value of the integral, since they fall in the interval $(1,4)$.

Exercise 16.1. Implement and compare two different algorithms for numerical integration of $\int_{x=0}^{1} x^2\mathrm{d}x$.

Solution: The most straightforward algorithm would be to sample X from the uniform distribution on $(0,1)$ and then use X^2 as an estimator.

Another algorithm, that is only slightly more work to implement, is to draw a random variable X with density $g(x) = 2x$ on $(0,1)$ and then use as an estimator $X/2$. Indeed, using that $f(x) = 1$ and $g(x) = 2x$, we find that the importance sampling estimator is given by

$$X^2\frac{f(X)}{g(X)} = \frac{1}{2}X, \tag{16.13}$$

and indeed

$$\mathbb{E}_g\left[\frac{1}{2}X\right] = \int_{x=0}^{1}\frac{1}{2}x2x\mathrm{d}x = \int_{x=0}^{1} x^2\mathrm{d}x. \tag{16.14}$$

Using the first algorithm yields an approximate 95% confidence interval 0.329 ± 0.018. The second algorithm yields an approximate 95% confidence interval 0.334 ± 0.007. The true value of the integral is $1/3$. Both intervals are based on 1000 independent runs.

Sample source code can be found in Listing 16.1.

16.3 Exponential Tilting

As stated in the previous section, it is usually hard to select an appropriate density g. For example, if our original density f is that of an exponential distribution, then the class of candidates for g are not only densities of all exponential distributions; but rather, it contains any density with support in $[0,\infty)$, such as the family of gamma densities, χ^2 densities and many other (non standard) densities.

It turns out to be useful to restrict ourselves to a certain parametrized family of candidates, known as the **exponential family generated by** X.

Listing 16.1: Two algorithms for estimating the value of $\int_{x=0}^{1} x^2 \mathrm{d}x$.

```
# Simulation 1, using uniform random variables
x = stats.uniform.rvs(0, 1, runs)
est1 = x**2

# Simulation 2, using exponential random variables
x = stats.uniform.rvs(0, 1, runs)
# If g(x) = 2x, then CDF G(x) = x^2. Sampling from this
# rv using inverse method: x = sqrt(y)
est2 = sqrt(x)/2

print(mean(est1))
print(var(est1))
print(mean(est2))
print(var(est2))
```

Definition 16.3. Let f be a density and let $M(\cdot)$ its associated moment generating function. Let $\Theta := \{\theta \in \mathbb{R} : M(\theta) < \infty\}$, then for $\theta \in \Theta$ define

$$f_\theta(x) := \frac{\mathrm{e}^{\theta x}}{M(\theta)} f(x). \tag{16.15}$$

then f_θ is called an **exponentially tilted density** and the family of all f_θ that can be associated to f is called the **exponential family generated by** f.

In many cases, the tilted density has the same parametric form as the original density. For example, in the case of the exponential distribution, the binomial distribution and the normal distribution.

Example 16.2. Suppose f is the density of the exponential distribution with rate λ, that is $f(x) = \lambda \mathrm{e}^{-\lambda x}$, $x \geq 0$. The associated moment generating function is $M(\theta) = \lambda/(\lambda - \theta)$, for $\theta < \lambda$. In this case

$$f_\theta(x) = \frac{\mathrm{e}^{\theta x}}{\frac{\lambda}{\lambda-\theta}} \lambda \mathrm{e}^{-\lambda x} = (\lambda - \theta)\mathrm{e}^{\theta x} \mathrm{e} - \lambda x = (\lambda - \theta)\mathrm{e}^{-(\lambda-\theta)x}, \tag{16.16}$$

which is the density of the exponential distribution with rate $\lambda - \theta$.

Example 16.3. Suppose f is the density of the normal distribution with parameters μ and σ^2, that is,

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}. \tag{16.17}$$

Then the associated moment generating function is $M(\theta) = \mathrm{e}^{\mu\theta + \theta^2\sigma^2/2}$ and in this case f_θ is given by

$$f_\theta(x) = \frac{\mathrm{e}^{\theta x}}{\mathrm{e}^{\mu\theta + \theta^2\sigma^2/2}} \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{1}{2\sigma^2}(x^2 - 2\mu x + \mu^2 - 2\theta x\sigma^2 + 2\theta\mu\sigma^2 + \theta^2\sigma^4)}. \tag{16.18}$$

Now since

$$x^2 - 2\mu x + \mu^2 - 2\theta x\sigma^2 + 2\theta\mu\sigma^2 + \theta^2\sigma^4 = (x - (\mu + \theta\sigma^2))^2, \tag{16.19}$$

it follows that

$$f_\theta(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \mathrm{e}^{-\frac{1}{2\sigma^2}(x - (\mu + \theta\sigma^2))^2}. \tag{16.20}$$

This is the density of the normal distribution with parameters $\mu + \theta\sigma^2$ and σ^2.

Example 16.4. Let f be the probability mass function of the binomial distribution with parameters n and p, that is, $f(k) = \binom{n}{k} p^k (1-p)^{n-k}$ for $k = 0, 1, \ldots, n$. The associated moment generating function is $M(\theta) = 1 - p + p\mathrm{e}^{\theta}$ and we find

$$f_\theta(k) = \binom{n}{k} p^k (1-p)^{n-k} \mathrm{e}^{\theta k} (1 - p + p\mathrm{e}^{\theta})^{-n} = \binom{n}{k} \left(\frac{p\mathrm{e}^{\theta}}{1 - p + p\mathrm{e}^{\theta}} \right)^k \left(\frac{1-p}{1 - p + p\mathrm{e}^{\theta}} \right)^{n-k} \tag{16.21}$$

and since

$$\frac{1-p}{1 - p + p\mathrm{e}^{\theta}} = \frac{1 - p + p\mathrm{e}^{\theta} - p\mathrm{e}^{\theta}}{1 - p + p\mathrm{e}^{\theta}} = 1 - \frac{p\mathrm{e}^{\theta}}{1 - p + p\mathrm{e}^{\theta}}, \tag{16.22}$$

it follows that

$$f_\theta(k) = \binom{n}{k} \left(\frac{p\mathrm{e}^{\theta}}{1 - p + p\mathrm{e}^{\theta}} \right)^k \left(1 - \frac{p\mathrm{e}^{\theta}}{1 - p + p\mathrm{e}^{\theta}} \right)^{n-k}, \tag{16.23}$$

which is the density of the binomial distribution with parameters n and $p\mathrm{e}^{\theta}/(1 - p + p\mathrm{e}^{\theta})$.

Lemma 16.1. *Let X be a random variable with density f and moment generating function $M_X(\cdot)$. Let Y be a random variable with density f_θ, then*

$$M_Y(\beta) = \frac{M_X(\theta + \beta)}{M_X(\theta)}. \tag{16.24}$$

Proof. The Lemma follows from straightforward calculation:

$$M_Y(\beta) = \int \mathrm{e}^{\beta y} f_\theta(y) \mathrm{d}y = \int \mathrm{e}^{\beta y} \frac{\mathrm{e}^{\theta y}}{M_X(\theta)} f(y) \mathrm{d}y = \frac{1}{M_X(\theta)} \int \mathrm{e}^{(\beta+\theta) y} f(y) \mathrm{d}y = \frac{M_X(\beta + \theta)}{M_X(\theta)}. \quad \square \tag{16.25}$$

Corollary 16.1. In the setting described in Lemma 16.1,

$$\mathbb{E}[Y] = \frac{M_X'(\theta)}{M_X(\theta)}. \tag{16.26}$$

Corollary 16.2. In the setting described above, let θ_0 be the solution to $M_X'(\theta) = 0$, then $\mathbb{E}[Y] \geq 0$ whenever $\theta \geq \theta_0$ and $\mathbb{E}[Y] \leq 0$ whenever $\theta \leq \theta_0$.

Suppose that $X_1, X_2, \ldots, X_n$ are i.i.d. random variables with common density f and we are interested in estimating

$$z = \mathbb{E}[h(X_1, X_2, \ldots, X_n)]. \tag{16.27}$$

Now suppose that we want to use importance sampling via exponential tilting, say we want to draw the X_i independently from f_θ instead. In this case, the likelihood ratio L takes a simple form,

$$L(Y_1, Y_2, \ldots, Y_n) = \prod_{i=1}^{n} \frac{f(Y_i)}{f_\theta(Y_i)} = \prod_{i=1}^{n} \frac{f(Y_i)}{\mathrm{e}^{\theta Y_i - \kappa(\theta)} f(Y_i)} = \mathrm{e}^{-\theta S_n + n\kappa(\theta)}, \tag{16.28}$$

where $\kappa(\cdot) = \log M(\cdot)$ is the cumulant generating function and $S_n = X_1 + X_2 + \ldots + X_n$. As an estimate for z, we can use

$$h(Y_1, Y_2, \ldots, Y_n) \mathrm{e}^{-\theta S_n + n\kappa(\theta)}. \tag{16.29}$$

The following example serves to illustrate the discussion above.

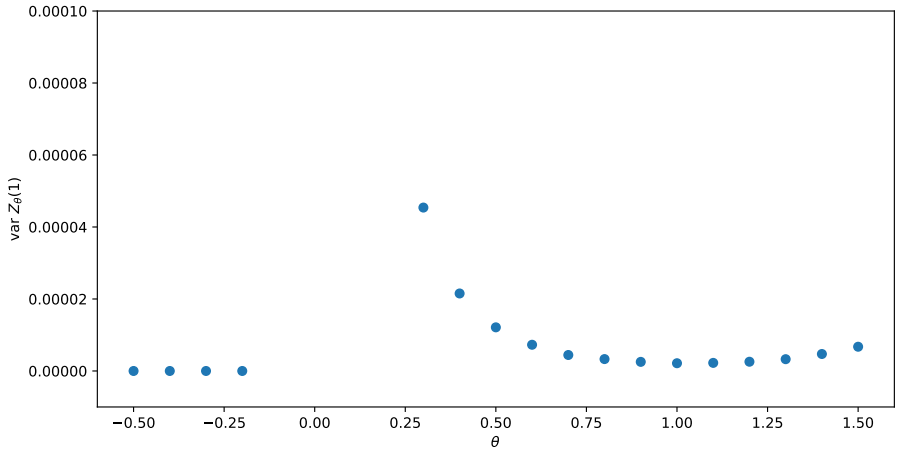


Figure 16.2: Estimated variance of $Z_\theta(1)$ in Example 16.5. Number of runs for each value of θ: 10000. We took $n = 10$ in all cases.

Example 16.5. Suppose that $X_1, X_2, \ldots, X_n$ are independent and are identically distributed according to the standard normal distribution and set $S_n := X_1 + X_2 + \ldots + X_n$. We are interested in estimating $z(x) = \mathbb{P}(S_n > nx)$, where $x > 0$, and we want to estimate $z(x)$ by importance sampling via exponential tilting. The crude monte carlo estimator is given by $Z_0(x) = \mathbf{1}_{\{S_n > nx\}}$ (in this case, $h(X_1, X_2, \ldots, X_n) = \mathbf{1}_{\{X_1+X_2+X_n > nx\}}$), but as $z(x) \to 0$ whenever $x \to \infty$, this estimator is not efficient.

The cumulant generating function is given by $\kappa(\theta) = \theta^2/2$. The exponentially tilted random variables $X_{\theta,1}, X_{\theta,2}, \ldots X_{\theta,n}$ follow a normal $(\theta, 1)$ distribution and the importance sampling estimator is given by

$$Z_\theta(x) = \mathbf{1}_{\{S_n > nx\}} \exp\left(-\theta S_{\theta,n} + \frac{n}{2}\theta^2\right), \qquad S_{\theta,n} = X_{\theta,1} + X_{\theta,2} + \ldots + X_{\theta,n}. \tag{16.30}$$

We plotted the estimated variance for $Z_\theta(x)$ for the case $x = 1, n = 10$ in Figure 16.2. We took $\theta \in \{-0.5, -0.4, \ldots, +1.5\}$ and for each value of θ, we repeated the simulation 10,000 times.

As is clear from the picture, for many values of $\theta < 0$, the estimated variance of the estimator is equal to 0. This is not because these estimators are so good, but the event $\{S_{\theta,n}/n > 1\}$ did not occur in any of those simulation runs (which should not come as a surprise, since $\mathbb{E}[S_{\theta,n}/n] = \theta < 0$.

It is also clear from the picture, that the optimal value of θ lies somewhere around $x = 1$, and it can be shown that this is no coincidence.

The source code used to generate this example can be found in Listing 16.2. To increase the speed, we used that the sum of n independent normal $(\theta, 1)$ random variables is, again, normally distributed, but with mean $n\theta$ and variance n.

So far, we replaced the problem of finding a nice importance sampling distribution within the space of all distribution functions, by the problem of selecting the best importance sampling estimator within the exponential family generated from the original distribution. However, an important question remains, namely, which value of θ should be chosen? Sometimes ad-hoc methods work well, as is illustrated in the following example. But often, finding the right value for theta requires a lot of hard work, as is showed later in this chapter.

Example 16.6. Suppose that $X_1, X_2, \ldots, X_n$ are i.i.d. standard normal random variables with moment generating function $M_X(\theta) = \mathrm{e}^{\theta^2/2}$. Let $S_n = X_1 + X_2 + \ldots + X_n$. Suppose that we want to estimate $z = \mathbb{P}(S_n \geq x)$, where $x > 0$. The crude Monte Carlo estimator is given by

$$Z = \mathbf{1}_{\{S_n \geq x\}}. \tag{16.31}$$

Listing 16.2: Python source code used to generate Figure 16.2.

```
    sTmp = stats.norm.rvs(n*theta, sqrt(n), size=runs)
    s = asarray(sTmp) # calculations below require a numpy array
    result = exp(0.5 * n * theta**2 - s * theta)
    result[s <= n * x] = 0
    return result

thetas = arange(-0.5, 1.6, 0.1)
x = 1
n = 10
runs = 10000

vars = [var(sim(x, n, th, runs)) for th in thetas]
plt.figure()
plt.plot(thetas, vars, 'o')
plt.ylim((-1E-5, 10E-5))
plt.show()
```

The importance sampling estimator is given by

$$Z_\theta = \mathbf{1}_{\{S_n \geq x\}} \prod_{i=1}^{n} \left(\mathrm{e}^{-\theta X_i} M_X(\theta) \right) = \mathbf{1}_{\{S_n \geq x\}} \mathrm{e}^{-\theta S_n} \left(M_X(\theta) \right)^n . \tag{16.32}$$

If $\theta > 0$, note that

$$0 \leq Z_\theta \leq \mathrm{e}^{-\theta x} \left(M_X(\theta) \right)^n . \tag{16.33}$$

Now using the fact that $M_X(\theta) = \mathrm{e}^{\theta^2/2}$, we find

$$0 \leq Z_\theta \leq \mathrm{e}^{-\theta x + n\theta^2/2}, \tag{16.34}$$

the right hand side of which is minimized as $-\theta x + n\theta^2/2$ is minimized, which is the case when $\theta = x/n$.

Exercise 16.2 (TP)**.** Show that for $\theta = x/n$ in the previous example, we have $\mathbb{E}\left[Z_\theta^2\right] \leq \exp\left(-\frac{x^2}{n}\right)$ and hence that $\mathrm{Var}\left[Z_n\right] \leq \exp\left(-\frac{x^2}{2n}\right) - z^2$.

Solution: The assertion follows from substituting $\theta = x/n$ into the estimate in Equation (16.34).

16.4 Random Walks and Queues

In this section, we will first describe a random walk with negative drift and derive an importance sampling estimator for the probability that such a random walk ever exceeds the value $x > 0$ ("hits the set $(x, +\infty)$" in the language of random walks). We will then give a connection of such random walks to the **steady state waiting time** in a queue.

Consider a sequence of i.i.d. random variables $X_1, X_2, \ldots$ with common density f. We will assume that the X_i are not concentrated on $(-\infty, 0]$, but that $\mathbb{E}[X_i] < 0$. Let us define $S_0 = 0$ and

$$S_n := \sum_{i=1}^{n} X_i, \qquad (n = 1, 2, 3, \ldots) \tag{16.35}$$

and

$$\tau(x) := \inf\{n : S_n > x\}, \qquad (x > 0). \tag{16.36}$$

The sequence of random variables S_n is known as a (discrete time) **random walk** on the real line. Since $\mathbb{E}[X_i] < 0$, it can be seen that $\mathbb{E}[X_{n+k}] < \mathbb{E}[X_n] < 0$ for all $n \geq 1$ and all $k > 1$.

The random variable $\tau(x)$ is called the hitting time of $(x, +\infty)$. It is, as the name suggests, the first time that the process S_n "hits" the set $(x, +\infty)$ (we will assume that $\tau(x) = \infty$ if $S_n \leq x$ for all n).

The quantity we are interested in is

$$z(x) := \mathbb{P}(\tau(x) < \infty), \tag{16.37}$$

the probability of ever hitting $(x, +\infty)$. It is clear from the definitions that $z(x) = 1$ if $x \leq 0$ and it can be shown that $z(x) < 1$ for $x > 0$ and $z(x) \downarrow 0$ if $x \to \infty$. This suggests that, if we choose x large, the event $\tau(x) < \infty$ will be a rare event.

Although at a first glance this setting may look similar to that in Example 16.5, it is actually more similar to the setting in Chapter 12, where we study the compound Poisson risk process. In fact, the problem sketched in Example 16.5 is much easier than the one sketched here. Since monte Carlo simulation in the example consists of drawing a fixed number of random variables, it is clear that each repetition of the simulation can be performed in finite time. In the problem in this section, the event $\{S_n > x \text{ for some } n\}$ need not occur, so some realizations of a CMC simulation will take infinite time to be completed. This shows that CMC is not only very hard because of the small probabilities, but is in fact impossible.

It turns out that importance sampling is not only fit to tackle the problems with rare events, but in this case also solves the problem with infinitely long simulation runs.

Assume that we apply importance sampling via exponential tilting, that is, we use as an estimator

$$Z_\theta(x) := \prod_{i=1}^{\tau(x)} \frac{f(X_i)}{f_\theta(X_i)} \mathbf{1}_{\{\tau(x)<\infty\}} = \mathrm{e}^{-\theta \sum_{i=1}^{\tau(x)} X_i + n\kappa(\theta)} \mathbf{1}_{\{\tau(x)<\infty\}}, \tag{16.38}$$

where the X_i have the tilted density f_θ.

Let $\theta = \theta_0$ be the solution to $M'(\theta) = 0$. By Corollary 16.2, $\mathbb{E}[f_\theta] X_i > 0$ when $\theta > \theta_0$. This can be shown to imply that the process crosses x with probability 1.

Let $\theta = \gamma > \theta_0$ be the solution of $M_X(\theta) = 1$ (we will assume that such a value γ exists). In this case the estimator $Z_\gamma(x)$ can be written as

$$Z_\gamma(x) = \mathrm{e}^{-\gamma S_n} \mathbf{1}_{\{\tau(x)<\infty\}} = \mathrm{e}^{-\gamma x} \mathrm{e}^{-\gamma \xi(x)}, \tag{16.39}$$

where $\xi(x)$ is the overshoot, defined as $\xi(x) := S_{\tau(x)} - x$.

It can be shown that $Z_\gamma(x)$ is an estimator of bounded relative error. Moreover, it is the unique importance sampling estimator with logarithmic efficiency (cf. [Asm07], Theorem 2.1 and Theorem 2.4).

Theorem 16.1. *The estimator $Z_\gamma(x) = e^{-\gamma x} e^{-\gamma \xi(x)}$ has bounded relative error and it is the unique importance sampling estimator with logarithmic efficiency.*

In probability theory, queues are mathematical abstractions of real-life queues. Customers arrive at the queue in a random fashion and are made to wait in line (the queue) until it is their time to be served. Service time is again random and when they are done, the customers leave the system.

Many variations of this subject exist, for example queues with multiple servers, servers with multiple lines, each of which has a different priority, or queues with a finite capacity (customers arriving when the queue is full are lost). We will now focus on a particular model, with a queue of infinite capacity, served by a single server. We will assume that the time between the arrival of the n-th and $(n+1)$-st customer (the so-called **interarrival time** T_n, $n = 0, 1, 2, ...$) has density f and moment generating function $M_T(\cdot)$, and the service time V_n ($n = 0, 1, 2, ...$) of the n-th customer has density g and moment generating function $M_V(\cdot)$. This model is known as the G/G/1 queue (the two G's describe the interarrival time distribution and the service time distribution, respectively; 'G' means 'General'; the number 1 indicates the number of servers).

One of the quantities in which probabilists are interested is the waiting time W_n of the n-th customer, for $n = 1, 2, ...$ (since it is assumed that the first customer arrives at time zero, we can set $W_0 = 0$). The time W_n describes the time between arrival and start of service of the n-th customer. If we let the system run for a long time, it may reach an equilibrium state; for example, the quantity W_∞ is known as the steady-state waiting time, which is the waiting time of a customer when the system is in equilibrium. Formally, W_∞ is the random variable with the limit distribution of W_n as $n \to \infty$ (provided that this limit exists, we will not go into details here).

Suppose that we are interested in the probability $\mathbb{P}(W_\infty > x)$ and that we want to estimate this quantity by means of simulation. In this section, we will relate this problem to the setting described above and show how this can be used to apply importance sampling to this problem.

We will start by defining quantities $X_n := V_{n-1} - T_{n-1}$ for $n = 1, 2, ...$, and $S_n = \sum_{i=1}^{n} X_i$. We will assume that the average service time is smaller than the average interarrival time. The practical implication of this assumption is that the system is *stable*, which means that the server is able to keep up with the arriving customers. Under this assumption, $\mathbb{E}[X_n] < 0$. It is reasonable to assume that X_n is not concentrated on $(-\infty, 0]$; which means that it may happen that the service time of a customer is longer than the time between this customer and the previous one.

If we define

$$\tau(x) := \inf\{n : S_n > x\} \tag{16.40}$$

as before, we are again in the setting as described above. In order to estimate $\mathbb{P}(\tau(x) < \infty)$, we can apply importance sampling via exponential twisting and the optimal parameter $\theta > \theta_0$ is given by the solution of $M_{X_n}(\theta) = 1$. Since V_{n-1} and T_{n-1} are independent, we find that

$$M_{X_n}(\theta) = M_V(\theta) M_T(-\theta). \tag{16.41}$$

At the moment, it is not clear that there exists a connection between $\mathbb{P}(\tau(x) < \infty)$ and $\mathbb{P}(W_\infty > x)$. However, we will show that these equalities are actually the same. In order to do so, note that

$$W_1 = \max(0, X_1) = \max(S_1 - S_1, S_1 - S_0); \tag{16.42}$$

$$\begin{aligned} W_2 &= \max(0, W_1 + X_2) \\ &= \max(S_2 - S_2, \max(S_1 - S_1, S_1 - S_0) + S_2 - S_1) \\ &= \max(S_2 - S_2, S_2 - S_1, S_2 - S_0), \end{aligned} \tag{16.43}$$

and it can be shown by induction that

$$W_n = \max_{k=0,1,...,n}(S_n - S_k). \tag{16.44}$$

We now have the following equality in distribution:

$$W_n = \max_{k=0,1,...,n}(S_n - S_k) \stackrel{\mathcal{D}}{=} \max_{k=0,1,...,n} S_k. \tag{16.45}$$

If we take $n \to \infty$ (provided, of course, that the limit exists), we find

$$W_\infty \overset{\mathcal{D}}{=} \max_{k=0,1,2,\ldots} S_k, \tag{16.46}$$

and since the events $\{\max_{k=0,1,2,\ldots} S_k > x\}$ and $\{\tau(x) < \infty\}$ are clearly equivalent, it follows that

$$\mathbb{P}(W_\infty > x) = \mathbb{P}(\tau(x) < \infty). \tag{16.47}$$

Example 16.7. In an M/M/1 queue, new customers arrive according to a Poisson process with intensity λ, so the interarrival times $T_0, T_1, T_2, \ldots$ follow an $\text{Exp}(\lambda)$ distribution and are independent. The service times $V_0, V_1, V_2, \ldots$ are i.i.d. $\text{Exp}(\mu)$. This means that the service times and interarrival times have moment generating function

$$M_T(\theta) = \frac{\lambda}{\lambda - \theta}, \qquad M_V(\theta) = \frac{\mu}{\mu - \theta}, \tag{16.48}$$

respectively. If we let $X_i = V_{i-1} - T_{i-1}$ and $S_k = X_1 + X_2 + \ldots + X_k$ in the discussion above, we find that

$$M_X(\theta) = M_V(\theta) M_T(-\theta) = \frac{\mu}{\mu - \theta} \frac{\lambda}{\lambda + \theta}, \tag{16.49}$$

and solving $M_X(\theta) = 1$ yields $\theta = \mu - \lambda$ as the unique positive solution, provided that $\mu > \lambda$.

Under the exponentially twisted distribution, the interarrival and service times follow an exponential distribution with parameter $\lambda + \theta = \mu$ and $\mu - \theta = \lambda$, respectively. It follows that in the unique asymptotically efficient importance sampling estimator, the arrival and service intensity are switched. If $\mu > \lambda$, this means that under the importance sampling distribution, the queue is unstable. As a consequence, the queue "blows up" and the event $\{\max_{k=0,1,2,\ldots} S_k > x\}$ is certain.

Exercise 16.3 (P)**.** Two pollen are placed on an infinite water surface, on which they move along a line. As have seen in Chapter 14, the movement of the pollen can be modeled by a so-called Brownian motion. For now, we will assume that the pollen move in discrete time, $t = 0, 1, 2, \ldots$, as follows. Call the respective pollen positions (at time t) P_t and Q_t, and let $k > 0$.

(i) At time 0, the pollen are in position k and $-k$, respectively, i.e. $P_0 = -k$ and $Q_0 = k$;

(ii) For every $t = 0, 1, 2$, we have that $P_{t+1} = P_t + X_t$ and $Q_{t+1} = Q_t + Y_t$. Here, X_t and Y_t are independent sequences of independent random variables, $X_t \sim \mathcal{N}(0, 1)$, and $Y_t \sim \mathcal{N}(1, 1)$.

Use simulation to estimate the probability that the pollen starting on the left will be at the right of the pollen that starts on the right at some point in time.

Solution: First, define the difference, $D_t := P_t - Q_t$. Note that D_t is a random walk with normally distributed increments,

$$D_t = -2k + \sum_{i=1}^{t} W_i, \qquad W_i = X_i - Y_i \sim \mathcal{N}(-1, 2) \text{ (i.i.d.)}. \tag{16.50}$$

Define τ to be the first time that the pollen switch their relative position, then τ is also the first time that D_t becomes nonnegative. We want to estimate the probability $\mathbb{P}(\tau < \infty)$. Note that the event $\tau = \infty$ has positive probability, since the pollen are drifting apart. Hence, crude Monte Carlo simulation will not work here.

The moment generating function of the increments W_i is $M_W(\theta) = \exp\left(-\theta + \theta^2\right)$, and the solutions of $M_W(\theta) = 1$ are $\theta = 0$ and $\theta = 1$, hence $\gamma = 1$. The twisted distribution of the increments is $\mathcal{N}(1,1)$, and we can use as an estimator

$$Z := \mathrm{e}^{-\sum_{i=1}^{\tau} W_i} \mathbf{1}_{\{\tau<\infty\}}. \tag{16.51}$$

See Listing 16.3 for a sample implementation.

Listing 16.3: Sample source code for estimating the probability that the pollen switch relative position.

```
k = 100
runs = 1000
theta = 1  # The IS estimator (should be > 0.5)

res = zeros(runs)
for i in range(runs):
    level = 0
    n = 0
    while level < 2*k:
        n += 1
        level += stats.norm.rvs(-1 + 2 * theta, 2, size=1)
    res[i] = exp(-theta * level)*exp(n*(-theta + theta**2))

m = mean(res)
halfWidth = 1.96 * sqrt(var(res)/runs)
ci = [m - halfWidth, m + halfWidth]
print(ci)
```

16.5 Importance sampling for the Compound Poisson Risk Process

In Chapter 12, we use Monte Carlo simulation to estimate the probability of eventual ruin $\psi(u)$, starting from a given surplus level u. The possibility that some of the simulation runs (in fact most of them) will go on forever (since ruin does not occur) proved to be troublesome. We tried to solve these problems by using approximate estimators instead, for instance by running the algorithm up to a fixed but large time t_∞.

In the discussion concerning queueing systems and random walks above, a similar problem arose. Namely, we tried to estimate the probability that the random walk S_n *ever* crosses the level x, which need not occur at all.

It turns out that estimating the probability of eventual ruin is not only similar to the setting described in the previous setting (estimation of $\mathbb{P}\left(\sum_{i=1}^{n} X_i > x\right)$, where the X_i are i.i.d., $x > 0$, and $\mathbb{E}[X_i] < 0$, but X_i is not concentrated on $(-\infty, 0]$), but actually falls within the same setting.

In order to show this, recall the formulation of the compound Poisson risk process,

$$\mathcal{U}(t) = u - \mathcal{Y}(t), \qquad \mathcal{Y}(t) = \sum_{i=1}^{N(t)} C_i - ct \tag{16.52}$$

where u is the insurer's surplus at time 0, the C_i are i.i.d. claims with mean μ, $N(t)$ is a Poisson process with intensity λ and c is the premium income rate, chosen such that $c = (1+\rho)\lambda\mu$, with $\rho > 0$. We are interested in $\mathbb{P}(\tau(u) < \infty)$, where $\tau(u) = \inf\{t \geq 0 : \mathcal{U}(t) < 0\} = \inf\{t \geq 0 : \mathcal{Y}(t) > u\}$.

If we denote by T_i the time between claim $i-1$ and claim i, the surplus level right after the arrival of the n-th claim can be written as

$$U(n) = u - Y(n), \qquad Y(n) = \sum_{i=1}^{n}(C_i - cT_i). \tag{16.53}$$

Like before, we let $\tau(u) = \inf\{n \geq 0 : U(n) < 0\} = \inf\{n \geq 0 : Y(n) > u\}$ and it is not hard to see that

$$\mathbb{P}(\tau(u) < \infty) = \mathbb{P}(\tau(u) < \infty). \tag{16.54}$$

Note that $Y(n)$ is of the form S_n in the previous section, and if we set $X_i = C_i - cT_i$, the correspondence between this model and the setting in the previous section becomes even more clear. To prove that our model fits in that setting, note that X_i is not concentrated on $(-\infty, 0]$ and

$$\mathbb{E}[X_i] = \mathbb{E}[C_i] - c\mathbb{E}[T_i] = \mu - c\frac{1}{\lambda} = \mu - (1+\rho)\lambda\mu\frac{1}{\lambda} < 0. \tag{16.55}$$

Example 16.8. Consider the compound Poisson risk model, in which claim sizes are distributed uniformly on $(0, 1)$. We want to apply importance sampling via exponential tilting to the estimation ultimate ruin.

Note that the moment generating functions of the claim size distribution and interarrival times are

$$M_C(\theta) = \frac{e^{\theta} - 1}{\theta}, \qquad M_T(\theta) = \frac{\lambda}{\lambda - \theta}, \tag{16.56}$$

respectively. In view of the discussion in the previous section, in order to find the optimal importance sampling parameter θ, we have to solve

$$1 = M_X(\theta) = M_C(\theta) M_T(-c\theta) = \frac{e^{\theta} - 1}{\theta}\frac{\lambda}{\lambda + c\theta}. \tag{16.57}$$

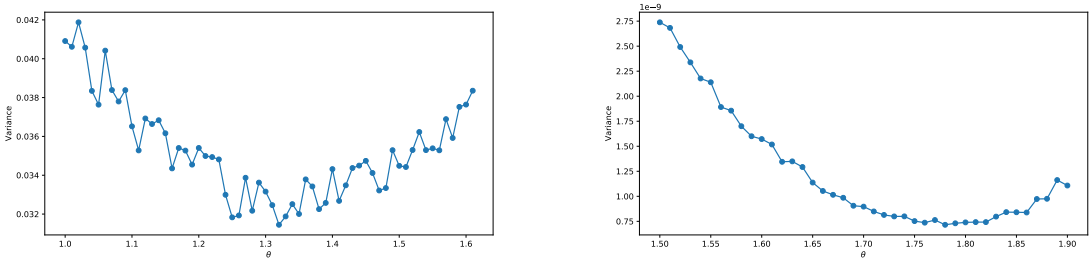
The solution of this equation can be found by numerical methods. For example, if we set $\lambda = 1$ and $c = 1$, we get $\theta \approx 1.79$.

In Table 16.1, estimates for $\psi(u)$ based on 1,000 independent simulation runs can be found for different values of u. These estimates are based on various simulation algorithms.

In Figure 16.3(a)-16.3(b), the estimated variance of the exponential tilting importance sampling estimator is plotted as a function of the tilting parameter θ. It can be seen that the optimal parameter $\theta \approx 1.79$ is only optimal for large values of u, while for $u = 0$ the value $\theta = 1.2$ is a little better.

	u	**Estimate**	**Variance**	**Half-width**
Beekman	0	0.5	0.2502503	0.03100583
	1	0.097	0.08767868	0.01835283
	5	0	0	0
	10	0	0	0
	100	0	0	0
IS, $\theta = 1.2$	0	0.4919172	0.03252397	0.01117784
	1	0.1034744	0.003826029	0.003833806
	5	8.256605e-05	1.201583e-08	6.794116e-06
	10	8.590206e-09	5.307469e-16	1.427907e-09
	20	1.91863e-16	1.504208e-30	7.601688e-17
IS, $\theta = 1.79$	0	0.4858517	0.04679667	0.01340799
	1	0.1013626	0.001203852	0.002150516
	5	7.967223e-05	7.185707e-10	1.661464e-06
	10	1.017382e-08	1.185417e-17	2.133987e-10
	20	1.671031e-16	2.953251e-33	3.368265e-18

Table 16.1: Simulation results for various approaches and for various initial capital levels. The rightmost column contains the half-width of the approximate 95% confidence interval. Every result is based on 1,000 independent runs. This table shows that the standard approach is useless for large values of u.



(a) $u = 0$ (b) $u = 5$

Figure 16.3: Plot of the variance of the exponential tilting importance sampling estimator as a function of the tilting parameter θ.

Bibliography

[And07] Anderson, D.F., *A modified next reaction method for simulating chemical systems with time dependent propensities and delays*, The Journal of Chemical Physics, 127, 2007.

[AG07] Asmussen, S. Glynn, P.W., *Stochastic Simulation: Algorithms and Analysis*, Springer, 2007.

[Bla73] Black F. and Scholes M., *The pricing of options and corporate liabilities*, Journal of Political Economy, 81(3):637-654, 1973.

[And99] Anderson E.C., *Monte Carlo Methods and Importance Sampling*, Lecture Notes for Stat 578C: Statistical Genetics, University of California, Berkeley, 1999. Via http://ib.berkeley.edu/labs/slatkin/eriq/classes/guest_lect/mc_lecture_notes.pdf.

[Asm07] Asmussen S. and Glynn P.W., *Stochastic Simulation, Algorithms and Analysis*, Springer, 2007.

[BOW03] Bluhm C., Overbeck L., Wagner C., *An Introduction to Credit Risk Modeling*, CRC Press, 2003.

[CHA87] Chandler, D., *Introduction to Modern Statistical Mechanics*, Oxford University Press, 1987.

[Chang99] Chang J., *Brownian Motion*, UC Santa Cruz, 1999. Available via http://users.soe.ucsc.edu/~chang/203/bm.pdf.

[Etheridge02] Etheridge A., *A Course in Financial Calculus*, Cambridge University Press, 2002.

[Kaas01] Kaas R., Goovaerts M., Dhaene J., Denuit M., *Modern Actuarial Risk Theory*, Kluwer Academic Publishers, 2001.

[MN98] Matsumoto M., Nishimura T., *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modeling and Computer Simulation 8 (1): 3–30, 1998

[Ross96] Ross S.M., *Stochastic Processes*, Wiley, 1996.

[Ross99] Ross S.M., *An Introduction to Mathematical Finance: Options and Other Topics*, Cambridge University Press, 1999.

[Ross07] Ross S.M., *Introduction to Probability Models*, Academic Press, 2007.

[WikiA] Wikipedia, *Student's t-distribution*, retrieved September 8, 2011, via http://en.wikipedia.org/wiki/Student's_t-distribution.

[WW02] Whitt, W., *Stochastic-Process Limits: An Introduction to Stochastic-Process Limits and Their Application to Queues*, Springer, 2002.

[Zanten10] Van Zanten H., *Introductory lectures on derivative pricing*, Eindhoven University of Technology, 2010.

Part V

Appendix

A

Short introduction to Python

There are many good Python introductions available on the web. The code below discusses many of the topics that are relevant for the stochastic simulations discussed in these lecture notes. It requires the following modules: numpy (array, sqrt, arange), statistics (mean, std, var), scipy (stats), matplotlib (pyplot), pandas (for data analysis).

Listing A.1: Introduction to Python with particular focus on MC simulation.

```
1
2  # You can use Python as a calculator
3  a = 2 + 3
4  b = 2 - 3
5  c = 14 / 5
6  c2 = 14 // 5   # floor division
7  d = 2 * -4
8  e = 19 % 4  # modulo
9  f = 4 ** 3 # power
10
11 print(a, b, c, c2, d, e, f)
12
13 # Working with variables
14 x = 5
15 y = x + 3
16 print(x, y)
17 x = 6
18 print(x, y)
19 z = x + y
20 print(z) # Result of sum
21
22 x = 3
23 x = x + 2 # increase the value of x
24 print(x)
25 x += 3
26 print(x)
27 x -= 2
28 print(x)
29
30 # you can use longer variable names
31 arrivalRate = 3.5
32 veryLongVariableName = 55
33 anotherLongOne = veryLongVariableName + 44
34
35 # Python has several solutions for arrays/vectors
36
37 # 1. Lists are the most commonly used type for arrays
38 v1 = [1, 2, 3, 4]
39 print(v1)
40 v2 = [5, 6, 7, 8]
41 # Note that the first index is 0:
42 print(v2[0])
43 # and negative numbers can be used to extract elements from end of list
44 print(v2[-1])
45
46 # You can use the "slice" operator (:)
```

```
47  # note that it includes the first element, but excludes the last argument
48  print(v2[0:3])
49
50  # you cannot do vector calculations on lists:
51  v3 = v1 + v2
52  print(v3)
53  v4 = [0] * 5
54  print(v4)
55
56  # several ways to add lists, but this is the most convenient
57  # note that range includes 0, but does not include the last element
58  vsum = [v1[i] + v2[i] for i in range(0, len(v1))]
59  print(vsum)
60
61
62
63  # although, if you want to do vector computations, it's probably
64  # better to use "arrays", defined in numpy
65  # numpy is a huge collection of numerical computations
66  from numpy import array
67  a1 = array(v1)
68  a2 = array(v2)
69  a3 = a1 + a2
70  print(a3)
71
72  # Other numpy functions also work on arrays,
73  # for example numpy.sqrt. math.sqrt would not allow this:
74  from numpy import sqrt
75  print(sqrt(a3))
76
77  # This is a nice feature, resulting in a boolean array:
78  b = a3 > 9
79  print(b)
80
81  # use this to select from a vector
82  print(a3[b])
83
84  # An efficient way to count elements satisfying some condition:
85  print(sum(a3 > 9))
86  print(sum(b)/len(b))
87
88
89
90  # you can remove or add elements to a list, but it's not very efficient
91  v5 = v3 + [11]
92  print(v5)
93  v3.append(11)
94  print(v3)
95  v3.pop()   # note that the LAST element is removed!
96  print(v3)
97  v3.append(12)
98  print(v3)
99  v3.pop(0)   # now the FIRST element is removed!
100 print(v3)
101
102 # you can change elements form a list:
103 v1[2] = 0.5
104 print(v1)
105
106
107 # An alternative to lists are "tuples". They are exactly like lists,
108 # but elements cannot be changed
109 t1 = 1, 2, 3, 4
110 t2 = 5, 6, 7, 8
111 t3 = t1, t2
112 print(t1)
113 print(t2)
114 print(t3)
115
116 print(t2[0])
117 # you cannot change elements of a tuple. This would give an error:
118 #t2[0] = 44
119
120 # Tuples are very useful if you have a function that returns multiple values
121
```

```
# A matrix can be implemented as a list of lists:
M1 = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(M1)
print(M1[1][2])
print(M1[1][1:2])
print(M1[1:3])
# But you cannot easily extract columns:
print(M1[0:3][1])  # probably does not return what you'd expect
# This is much easier if you use numpy arrays:
M1a = array(M1)
print(M1a)
print(M1a[0:3,1])  # Note the slightly different syntax.
print(M1a[0:3,1:2])  # This is not possible with lists
print(M1a[0:3,1:3])


# numpy is very suitable for matrix computations:
from numpy import arange
nM1 = array(M1)
nM2 = arange(9).reshape(3, 3)
print(nM2)
print(nM1 + nM2)
# Matrix multiplication
print(nM1.dot(nM2))
# Transpose
print(nM1.transpose())


# Functions
def square(x):
    return x**2

print(square(3.5))

def multiply(a, b):
    return a*b

print(multiply(4.5, -2))


# The for-loop. Note that 10 is not included in the range!
s = 0
for i in range(1, 10) :
    s += i

print(s)

print(v1)  # v1 = [1, 2, 0.5, 4]
s = 0
for i in range(0, len(v1)) :
    s += v1[i]
print(s)

# This is more elegant:
s = 0
for i in v1 :
    s += i
print(s)

# If you don't know how many iterations you will need,
# the while loop is more suitable.
i = 0
s = 0
while i < len(v1) :
    s = s + v1[i]
    i = i + 1
print(s)



# A first stochastic simulation example:
# Suppose you throw a fair die until you reach a total of
# at least 50 points. How many times do you have to throw?
# And what is the total number of points you end up with?
import random
```

```
197
198 # sample 10 random numbers from 1, 2, ..., 6:
199 for i in range(1, 10) :
200     print(random.randint(1, 6))
201
202 # You have to be extremely careful, because there are
203 # two different random classes (the standard one we used
204 # above, and one in numpy). They sometimes have different
205 # behaviour, in particular with the randint function!
206 import numpy as np
207 from numpy import mean
208
209 sim1 = [random.randint(1, 6) for _ in range(1000000)]
210 sim2 = [np.random.randint(1, 6) for _ in range(1000000)]
211 print('random: ')
212 print(max(sim1), mean(sim1))  # includes the 6
213 print('numpy random: ')
214 print(max(sim2), mean(sim2))  # does NOT include sixes!!!
215
216 # Conclusion: in Python there are multiple libraries to sample
217 # random numbers. We will use the numpy library in our lectures.
218 # The numpy 'random' function is deprecated. Use random
219 # number generators instead, see also Chapter 3.
220 from numpy import random
221
222 rng = random.default_rng()   # The random number generator
223
224
225 # So be careful, now we're using numpy.random:
226 print('numpy random (again): ')
227 sim2 = [rng.integers(1, 7) for _ in range(1000000)]  # note the 7!
228 print(max(sim2), mean(sim2))  # now includes sixes!!!
229
230 # One game
231 counter = 0
232 total = 0
233 while total < 50 :
234     total += rng.integers(1, 7)
235     counter += 1
236
237 print(counter, total)
238
239 # Now do many, many runs:
240 nrRuns = 100000
241 counterValues = [0]*nrRuns
242 totalValues = [0]*nrRuns
243 for i in range(0, nrRuns) :
244     counter = 0
245     total = 0
246     while total < 50 :
247         total += rng.integers(1, 7)
248         counter += 1
249     counterValues[i] = counter
250     totalValues[i] = total
251
252 # Python contains a module with functionality
253 # for probability distributions
254 from statistics import mean
255 print(mean(counterValues))
256 print(mean(totalValues))
257
258 # the if statement:
259 a = 7
260 if a > 5 :
261     print('Greater than 5')
262 else :
263     print('Smaller than, or equal to, 5')
264
265 # ADVANCED TOPIC: Dictionaries - WILL NOT BE DISCUSSED IN THIS COURSE
266 # Now find the distribution. A dictionary is extremely useful here
267 # This example is from the "Think Python" tutorial
268 def histogram(s):
269     d = dict()
270     for c in s:
271         if c not in d:
```

```
            d[c] = 1
        else:
            d[c] += 1
    return d

hist = histogram(totalValues)
# we know that the only possible outcomes are 50, 51, ..., 55:
for value in range(50, 56) :
    print(value, hist[value]/nrRuns)

hist2 = histogram(counterValues)
# we know that the only possible outcomes are 9, 10, ..., 50:
for value in range(9, 51) :
    if value in hist2 :
        print(value, hist2[value]/nrRuns)
    else :
        print(value, 0)
# END ADVANCED TOPIC


# The library 'pandas' is excellent for data analysis
import pandas

pcv = pandas.Series(counterValues)
pcv.value_counts(sort=False, normalize=True)


# Example: a function that computes the maximum of two numbers
def maximum(a, b):
    if a > b :
        return a
    else :
        return b

print(maximum(5, -3))

# Example: a function that computes the maximum of three numbers
def maximum3(a, b, c):
    if a > b :
        if a > c :
            return a
        else :
            return c
    elif b > c :
        return b
    else :
        return c

print(maximum3(5, -5, 3))

# Exercise: write a function that computes the maximum of a list of numbers
import math

def maxlist(vec) :
    mx = -math.inf
    for x in vec :
        if x > mx :
            mx = x
    return mx

print(maxlist([4, 7, -22, 3.3, 0, 11, -1111]))


# The "AND" and "OR" operators
player1 = rng.integers(1, 7)
player2 = rng.integers(1, 7)
print(player1, player2)
if player1 == 6 and player2 == 6 :
    print('Both players threw 6')
elif player1 == 6 or player2 == 6 :
    print('One of the players threw 6')
else :
    print('None of the players threw 6')

# The "NOT" operator
```

```
347 if player1 == 6 and not player2 == 6 :
348     print('Only player 1 threw 6')
349
350
351 # Using random variables
352 from scipy import stats
353 expDist = stats.expon(scale=3.0)    # note that this is the MEAN!!!!
354 normDist = stats.norm(3.0, 2.0)  # scale = 2.0 means standard deviation = 2
355 n = 100000
356 sampleExp = expDist.rvs(n)
357 sampleNorm = normDist.rvs(n)
358
359 # Creating plots
360 import matplotlib.pyplot as plt
361
362 # ECDF. Note the numpy function arange, because range only works with integers
363 ys = arange(1/n, 1+1/n, 1/n)
364 xs1 = sorted(sampleExp)
365 xs2 = sorted(sampleNorm)
366 plt.figure()
367 plt.step(xs1, ys, color='green')
368 plt.step(xs2, ys, color='blue')
369
370
371 # Histogram exponential distribution
372 plt.figure()   # create a new plot window
373 plt.hist(sampleExp, bins=100, rwidth=0.8, density=True)
374 # Add theoretical density
375 xs = arange(min(sampleExp), max(sampleExp), 0.1)
376 ys = expDist.pdf(xs)
377 plt.plot(xs, ys, color="red")
378
379 # Histogram normal distribution
380 plt.figure(figsize=(8,5))
381 plt.hist(sampleNorm, bins=100, rwidth=0.8, density=True)
382 xs = arange(min(sampleNorm), max(sampleNorm), 0.1)
383 ys = normDist.pdf(xs)
384 plt.plot(xs, ys, color="red")
385 plt.show() # show all plot windows
```

B

Efficiency of MC simulation in Python

B.1 Sampling random numbers

Our experience is that Python is comparable in speed to programming languages like R and Matlab, but slower than C++ and Java. In our attempt to speed up our code, we noticed one particular function that caused our programs to be much slower than we expected. This was the `rvs()` function in the statistical distributions from the `scipy.stats` module. To be more precise, this function is relatively slow when sampling one random number at a time. On the contrast, it is very fast when sampling large vectors of random numbers. Still, in some of our simulations we cannot avoid having to sample one number at a time (think of the service times in Chapters 8 and 9. For this reason, we have developed a class called `Distribution` that can be used like the regular distribution classes, but it samples large number of random numbers at once. This significantly speeds up simulations and we recommend using it whenever the `rvs()` function is used to sample small numbers of random variates. To increase the compatibility with distribution objects from the `scipy.stats` module, we also added common functions like `mean`, `std`, `cdf`, and so on. The code can be found in Listing B.1. A program that actually compares the speed of sampling with and without the Distribution class is shown in Listing B.2.

We also want to warn you for unexpected behaviour when sampling random numbers using the `random` class. You should realise that there are *two* modules called random: one in the "standard" library, and one in `numpy`. They sometimes have different behaviour, most notably in their `randint` function. The "standard" `random` has a function `randint(a, b)` that includes the upper bound b, whereas the `numpy` version of random does *not* include this upper bound b and samples numbers $a, a+1, \ldots, b-1$. If you look at the Python introduction in Listing A.1 lines 202–212, you will see an example where these differences are highlighted.

B.2 Adding and removing elements from a list

We have used the double-ended queue for lists where elements are only (or mostly) added and removed at the start or the end. To get an impression of the speed gained with this implementation, we compared the `deque` with ordinary lists in Python. See Listing B.3.

Listing B.1: The Distribution class, created to speed up sampling random variates.

```
class Distribution :

    n = 10000 # standard random numbers to generate

    '''
    Constructor for this Distribution class.

    Args:
            dist (scipy.stats random variable): A random variable from
            the scipy stats libary.

    Attributes:
            dist (scipy.stats random variable): A random variable from
            the scipy stats libary.
            n (int): a number indicating how many random numbers should
            be generated in one batch
            randomNumbers: a list of n random numbers generated from 'dist'
            idx (int): a number keeping track of how many random numbers
            have been sampled

    '''

    def __init__(self, dist):
        self.dist = dist
        self.resample()

        '''
        Sets the random state of the (internal) random number generator.
        This is typically used to have control over the random seeds.
        '''

    def setRandomState(self, rng):
        self.dist.random_state = rng
        self.resample()


    def __str__(self):
        return str(self.dist)

    def resample(self):
        self.randomNumbers = self.dist.rvs(self.n)
        self.idx = 0

    def rvs(self, n=1):
        '''
        A function that returns n (=1 by default) random numbers from
        the specified distribution.

        Returns:
            One random number (float) if n=1, and a list of n random numbers
            otherwise.
        '''
        if self.idx >= self.n - n :
            while n > self.n :
                self.n *= 10
            self.resample()
        if n == 1 :
            rs = self.randomNumbers[self.idx]
        else :
            rs = self.randomNumbers[self.idx:(self.idx+n)]
        self.idx += n
```

Listing B.2: A program to test the speed of different implementations.

```
n = 100000

mu = 3.4
sigma = 1.5
t1 = time.time()
for i in range(n) :
    random.gauss(mu, sigma)
t2 = time.time()
print('random, normal distribution, 1 number at a time: %f sec' % (t2 - t1))

normDist = stats.norm(mu, sigma)
t1 = time.time()
for i in range(n) :
    normDist.rvs()
t2 = time.time()
print('stats rvs, normal distribution, 1 number at a time: %f sec' % (t2 - t1))

t1 = time.time()
normDist.rvs(n)
t2 = time.time()
print('stats rvs, normal distribution, n numbers at a time: %f sec' % (t2 - t1))

normDist = stats.norm(mu, sigma)
myDist = Distribution(normDist)
t1 = time.time()
for i in range(n) :
    myDist.rvs()
t2 = time.time()
print('Distribution, 1 number at a time: %f sec' % (t2 - t1))

t1 = time.time()
myDist.rvs(n)  # might take zero time, because sample was created in constructor!
t2 = time.time()
print('Distribution, n numbers at a time: %f sec' % (t2 - t1))
```

Listing B.3: A program to test the speed of different implementations.

```
n = 100000
v = []
t1 = time.time()
for i in range(n):
    v.append(i)
for i in range(n):
    del(v[0])
t2 = time.time()
print('list elements adding and removing specific item: %f sec' % (t2 - t1))

v = []
t1 = time.time()
for i in range(n):
    v.append(i)
for i in range(n):
    v.pop()
t2 = time.time()
print('list elements adding and removing item at index: %f sec' % (t2 - t1))

v = deque()
t1 = time.time()
for i in range(n):
    v.append(i)
for i in range(n):
    v.remove(v[0])
t2 = time.time()
print('deque elements adding and removing specific item: %f sec' % (t2 - t1))

v = deque()
t1 = time.time()
for i in range(n):
    v.append(i)
for i in range(n):
    v.pop()
t2 = time.time()
print('deque elements adding and popping: %f sec' % (t2 - t1))
```