Half Title Page

Title Page

LOC Page

Vince: to Riggins
Geraint: also, to Riggins

Contents

Section III Dynamical Systems

Chapter   4 ▪ Modelling with Differential Equations   47

Foreword

This is the foreword

Preface

This is the preface.

Contributors

Michaél Aftosmis
NASA Ames Research Center
Moffett Field, California

Pratul K. Agarwal
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Sadaf R. Alam
Oak Ridge National Laboratory
Oak Ridge, Tennessee

Gabrielle Allen
Louisiana State University
Baton Rouge, Louisiana

Martin Sandve Alnæs
Simula Research Laboratory and University
   of Oslo, Norway
Norway

Steven F. Ashby
Lawrence Livermore National Laboratory
Livermore, California

David A. Bader
Georgia Institute of Technology
Atlanta, Georgia

Benjamin Bergen
Los Alamos National Laboratory
Los Alamos, New Mexico

Jonathan W. Berry
Sandia National Laboratories
Albuquerque, New Mexico

Martin Berzins
University of Utah

Salt Lake City, Utah

Abhinav Bhatele
University of Illinois
Urbana-Champaign, Illinois

Christian Bischof
RWTH Aachen University
Germany

Rupak Biswas
NASA Ames Research Center
Moffett Field, California

Eric Bohm
University of Illinois
Urbana-Champaign, Illinois

James Bordner
University of California, San Diego
San Diego, California

Geörge Bosilca
University of Tennessee
Knoxville, Tennessee

Grèg L. Bryan
Columbia University
New York, New York

Marian Bubak
AGH University of Science and Technology
Kraków, Poland

Andrew Canning
Lawrence Berkeley National Laboratory
Berkeley, California

Jonathan Carter
Lawrence Berkeley National Laboratory
Berkeley, California

Zizhong Chen
Jacksonville State University
Jacksonville, Alabama

Joseph R. Crobak
Rutgers, The State University of New
    Jersey

Piscataway, New Jersey

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

I

Getting Started

Introduction

T HANK you for starting to read this book. This book aims to bring together two fascinating topics:

- Problems that can be solved using mathematics;

- Software that is free to use and change.

What we mean by both of those things will become clear through reading this chapter and the rest of the book.

1.1   WHO IS THIS BOOK FOR?

Anyone who is interested in using mathematics and computers to solve problems will hopefully find this book helpful.

If you are a student of a mathematical discipline, a graduate student of a subject like operational research, a hobbyist who enjoys solving the travelling salesman problem or even if you get paid to do this stuff: this book is for you. We will introduce you to the world of open source software that allows you to do all these things freely.

If you are a student learning to write code, a graduate student using databases for their research, an enthusiast who programmes applications to help coordinate the neighbourhood watch, or even if you get paid to write software: this book is for you. We will introduce you to a world of problems that can be solved using your skill sets.

It would be helpful for the reader of this book to:

- Have access to a computer and be able to connect to the internet (at least once) to be able to download the relevant software.

- Be prepared to read some mathematics. Technically you do not need to understand the specific mathematics to be able to use the tools in this book. The topics covered use some algebra, calculus and probability.

1.2   WHAT DO WE MEAN BY APPLIED MATHEMATICS?

We consider this book to be a book on applied mathematics. This is not however a universal term, for some applied mathematics is the study of mechanics and involves

modelling projectiles being fired out of canons. We will use the term a bit more freely here and mean any type of real world problem that can be tackled using mathematical tools. This is sometimes referred to as operational research, operations research, mathematical modelling or indeed just mathematics.

One of the authors, Vince, used mathematics to plan the sitting plan at his wedding. Using a particular area of mathematics call graph theory he was able to ensure that everyone sat next to someone they liked and/or knew.

The other author, Geraint, used mathematics to find the best team of Pokemon. Using an area of mathematics call linear programming which is based on linear algebra he was able to find the best makeup of pokemon.

Here, applied mathematics is the type of mathematics that helps us answer questions that the real world asks.

1.3 WHAT IS OPEN SOURCE SOFTWARE

Strictly speaking open source software is software with source code that anyone can read, modify and improve. In practice this means that you do not need to pay to use it which is often one of the first attractions. This financial aspect can also be one of the reasons that someone will not use a particular piece of software due to a confusion between cost and value: if something is free is it really going to be any good?

In practice open source software is used all of the world and powers some of the most important infrastructure around. For example, one should never use any cryptographic software that is not open source: if you cannot open up and read things than you should not trust it (this is indeed why most cryptographic systems used are open source).

Today, open source software is a lot more than a licensing agreement: it is a community of practice. Bugs are fixed faster, research is implemented immediately and knowledge is spread more widely thanks to open source software. Bugs are fixed faster because anyone can read and inspect the source code. Most open source software projects also have a clear mechanisms for communicating with the developers and even reviewing and accepting code contributions from the general public. Research is implemented immediately because when new algorithms are discovered they are often added directly to the software by the researchers who found them. This all contributes to the spread of knowledge: open source software is the modern should of giants that we all stand on.

Open source software is software that, like scientific knowledge is not restricted in its use.

1.4 HOW TO GET THE MOST OUT OF THIS BOOK

The book itself is open source. You can find the source files for this book online at `github.com/drvinceknight/ampwoss`. There will will also find a number of *Jupyter notebooks* and *R markdown files* that include code snippets that let you follow along.

We feel that you can choose to read the book from cover to cover, writing out

the code examples as you go; or it could also be used as a reference text when faced with particular problem and wanting to know where to start.

The book is made up of 10 chapters that are paired in two 4 parts. Each part corresponds to a particular area of mathematics, for example "Emergent Behaviour". Two chapters are paired together for each chapter, usually these two chapters correspond to the same area of mathematics but from a slightly different scale that correspond to different ways of tackling the problem.

Every chapter has the following structure:

1. Introduction - a brief overview of a given problem type. Here we will describe the problem at hand in general terms.

2. An Example problem. This will provide a tangible example problem that offers the reader some intuition for the rest of the discussion.

3. Solving with Python. We will describe the mathematical tools available to us in a programming language called Python to solve the problem.

4. Solving with R. Here we will do the same with the R programming language.

5. Brief theoretic background with pointers to reference texts. Some readers might like to delve in to the mathematics of the problem a bit further, we will include those details here.

6. Examples of research using these methods. Finally, some readers might even be interested in finding out a bit more of what mathematicians are doing on these problems. Often this will include some descriptions of the problem considered but perhaps at a much larger scale than the one presented in the example.

For a given reader, not all sections of a chapter will be of interest. Perhaps a reader is only interested in R and finding out more about the research. Please do take from the book what you find useful.

II

Probabilistic Modelling

Markov Chains

M ANY real world situations have some level of unpredictability through random-ness: the flip of a coin, the number of orders of coffee in a shop, the winning numbers of the lottery. However, mathematics can in fact let us make predictions about what we expect to happen. One tool used to understand randomness is Markov chains, an area of mathematics sitting at the intersection of probability and linear algebra.

2.1   PROBLEM

Consider a barber shop. The shop owners have noticed that customers will not wait if there is no room in their waiting room and will choose to take their business elsewhere. The Barber shop would like to make an investment so as to avoid this situation. They know the following information:

- They currently have 2 barber chairs (and 2 barbers).

- They have waiting room for 4 people.

- They usually have 10 customers arrive per hour.

- Each Barber takes about 15 minutes to serve a customer so they can serve 4 customers an hour.

This is represented diagrammatically in Figure 2.1.

They are planning on reconfiguring space to either have 2 extra waiting chairs or another barber's chair and barber.

The mathematical tool used to model this situation is a Markov chain.

2.2   THEORY

A Markov chain is a model of a sequence of random events that is defined by a collection of **states** and rules that define how to move between these states.

For example, in the barber shop a single number is sufficient to describe the status of the shop. If that number is 1 this implies that 1 customer is currently having their
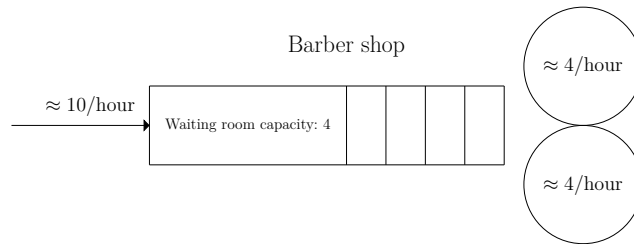
Figure 2.1 Diagrammatic representation of the barber shop as a queuing system.

hair cut. If that number is 5 this implies that 2 customers are being served and 3 are waiting. The entire state space is, in this case a finite set of integers from 0 to 6. If the system is full (all barbers and waiting room occupied) then we are in state 6 and if there is no one at the shop then we are in state 0. This is denoted mathematically as:

$$S = \{0, 1, 2, 3, 4, 5, 6\} \tag{2.1}$$

As customers arrive and leave the system goes between states as shown in Figure 2.2.
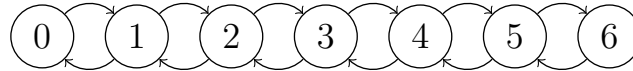
Figure 2.2 Diagrammatic representation of the state space

The rules that govern how to move between these states can be defined in two ways:

- Using probabilities of changing state (or not) in a well defined time period. This is called a discrete Markov chain.

- Using rates of change from one state to another. This is called a continuous time Markov chain.

For our barber shop we will consider it as a continuous Markov chain as shown in Figure 2.3

Note that a Markov chain assumes the rates follow an exponential distribution. One interesting property of this distribution is that it is considered memoryless which means that if a customer has been having their hair cut for 5 minutes this does not change the rate at which their service ends. This distribution is quite common in the real world and therefore a common assumption.

These states and rates can be represented mathematically using a transition matrix Q where Q_{ij} represents the rate of going from state i to state j. In this case we have:

Figure 2.3 Diagrammatic representation of the state space and the transition rates

$$Q = \begin{pmatrix} -10 & 10 & 0 & 0 & 0 & 0 & 0 \\ 4 & -14 & 10 & 0 & 0 & 0 & 0 \\ 0 & 8 & -18 & 10 & 0 & 0 & 0 \\ 0 & 0 & 8 & -18 & 10 & 0 & 0 \\ 0 & 0 & 0 & 8 & -18 & 10 & 0 \\ 0 & 0 & 0 & 0 & 8 & -18 & 10 \\ 0 & 0 & 0 & 0 & 0 & 8 & -8 \end{pmatrix} \tag{2.2}$$
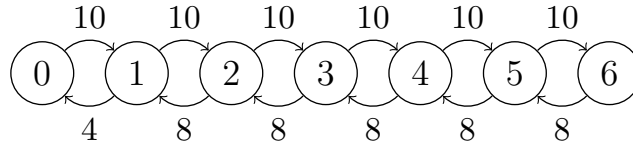
You will see that Q_{ii} are negative and ensure the rows of Q sum to 0. This gives the total rate of change leaving state i.

We can use Q to understand the probability of being in a given state after t time unis. This is can be represented mathematically using a matrix P_t where $(P_t)_{ij}$ is the probability of being in state j after t time units having started in state i. We can use Q to calculate P_t using the matrix exponential:

$$P_t = e^{Qt} \tag{2.3}$$

What is also useful is understanding the long run behaviour of the system. This allows us to answer questions such as "what state are we most likely to be in on average?" or "what is the probability of being in the last state on average?".

This long run probability distribution over the state can be represented using a vector π where π_i represents the probability of being in state i. This vector is in fact the solution to the following matrix equation:

$$\pi Q = 0 \tag{2.4}$$

In the upcoming sections we will demonstrate all of the above concepts.

2.3 SOLVING WITH PYTHON

The first step we will take is to write a function to obtain the transition rates between two given states:

Python input

```python
def get_transition_rate(
    in_state, out_state, waiting_room=4, num_barbers=2,
):
    """Return the transition rate for two given states.

    Args:
        in_state: an integer
        out_state: an integer
        waiting_room: an integer (default: 4)
        num_barbers:  an integer (default: 2)

    Returns:
        A real.
    """
    arrival_rate = 10
    service_rate = 4

    capacity = waiting_room + num_barbers
    delta = out_state - in_state

    if delta == 1 and in_state < capacity:
        return arrival_rate

    if delta == -1:
        return min(in_state, num_barbers) * service_rate

    return 0
```

Next, we write a function that creates an entire transition rate matrix Q for a given problem. We will use the numpy to handle all the linear algebra and the itertools library for some iterations:

Python input

```python
28  import itertools
29  import numpy as np
30
31
32  def get_transition_rate_matrix(waiting_room=4, num_barbers=2):
33      """Return the transition matrix Q.
34
35      Args:
36          waiting_room: an integer (default: 4)
37          num_barbers: an integer (default: 2)
38
39      Returns:
40          A matrix.
41      """
42      capacity = waiting_room + num_barbers
43      state_pairs = itertools.product(
44          range(capacity + 1), repeat=2
45      )
46
47      flat_transition_rates = [
48          get_transition_rate(
49              in_state=in_state,
50              out_state=out_state,
51              waiting_room=waiting_room,
52              num_barbers=num_barbers,
53          )
54          for in_state, out_state in state_pairs
55      ]
56      transition_rates = np.reshape(
57          flat_transition_rates, (capacity + 1, capacity + 1)
58      )
59      np.fill_diagonal(
60          transition_rates, -transition_rates.sum(axis=1)
61      )
62
63      return transition_rates
```

Using this we can obtain the matrix Q for our default system:

<div align="center">Python input</div>

```
64  Q = get_transition_rate_matrix()
65  print(Q)
```

which gives:

<div align="center">Python output</div>

```
66  [[-10  10   0   0   0   0   0]
67   [  4 -14  10   0   0   0   0]
68   [  0   8 -18  10   0   0   0]
69   [  0   0   8 -18  10   0   0]
70   [  0   0   0   8 -18  10   0]
71   [  0   0   0   0   8 -18  10]
72   [  0   0   0   0   0   8  -8]]
```

We can take the matrix exponential as discussed above. To do this, we need to use the `scipy` library. To see what would happen after .5 time units we obtain:

<div align="center">Python input</div>

```
73  import scipy.linalg
74
75  print(scipy.linalg.expm(Q * 0.5).round(5))
```

which gives:

<div align="center">Python output</div>

```
76  [[0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.0815 ]
77   [0.08501 0.18292 0.18666 0.1708  0.14377 0.1189  0.11194]
78   [0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346]
79   [0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142]
80   [0.02667 0.07361 0.10005 0.13422 0.17393 0.2189  0.27262]
81   [0.01567 0.0487  0.07552 0.11775 0.17512 0.24484 0.32239]
82   [0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914]]
```

To see what would happen after 500 time units we obtain:

```
     ┌─ Python input ─────────────────────────────────────────────┐
     │                                                             │
     │                                                             │
  83 │ print(scipy.linalg.expm(Q * 500).round(5))                  │
     │                                                             │
     └─────────────────────────────────────────────────────────────┘
```

which gives:

```
     ┌─ Python output ────────────────────────────────────────────┐
     │                                                             │
  84 │ [[0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]   │
  85 │  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]   │
  86 │  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]   │
  87 │  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]   │
  88 │  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]   │
  89 │  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]   │
  90 │  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]]  │
     │                                                             │
     └─────────────────────────────────────────────────────────────┘
```

We see that no matter what state (row) the system is in, after 500 time units the probabilities are all the same. We could in fact stop our analysis here, however our choice of 500 time units was arbitrary and might not be the correct amount for all possible scenarios, as such we will continue to aim to solve the underlying equation 2.4 directly.

To do this we will solve the underlying system using a numerically efficient algorithm called least squares optimisation (available from the `numpy` library):

Python input

```python
 91  def get_steady_state_vector(Q):
 92      """Return the steady state vector of any given continuous
 93      time transition rate matrix.
 94
 95      Args:
 96          Q: a transition rate matrix
 97
 98      Returns:
 99          A vector
100      """
101      state_space_size, _ = Q.shape
102      A = np.vstack((Q.T, np.ones(state_space_size)))
103      b = np.append(np.zeros(state_space_size), 1)
104      x, _, _, _ = np.linalg.lstsq(A, b, rcond=None)
105      return x
```

So if we now see the steady state vector for our default system:

Python input

```python
106  print(get_steady_state_vector(Q).round(5))
```

we get:

Python output

```
107  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094  0.26176]
```

We can see that the shop is expected to be empty approximately 3.4% of the time and full 26.2% of the time.

The final function we will write is one that uses all of the above to just return the probability of the shop being full.

```
─────────────────────────── Python input ───────────────────────────
108  def get_probability_of_full_shop(
109      waiting_room=4, num_barbers=2
110  ):
111      """Return the probability of the barber shop being full.
112
113      Args:
114          waiting_room: an integer (default: 4)
115          num_barbers: an integer (default: 2)
116
117      Returns:
118          A real.
119      """
120      Q = get_transition_rate_matrix(
121          waiting_room=waiting_room, num_barbers=num_barbers,
122      )
123      pi = get_steady_state_vector(Q)
124      return pi[-1]
```

We can now confirm the previous probability calculated probability of the shop being full:

```
─────────────────────────── Python input ───────────────────────────
125  print(round(get_probability_of_full_shop(), 6))
```

which gives:

```
─────────────────────────── Python output ───────────────────────────
126  0.261756
```

If we were too have 2 extra space in the waiting room:

```
─────────────────────────── Python input ───────────────────────────
127  print(round(get_probability_of_full_shop(waiting_room=6), 6))
```

which gives:

```
──── Python output ────
0.23557
```
128

This is a slight improvement however, increasing the number of barbers has a substantial effect:

```
──── Python input ────
print(round(get_probability_of_full_shop(num_barbers=3), 6))
```
129

```
──── Python output ────
0.078636
```
130

2.4 SOLVING WITH R

The first step we will take is write a function to obtain the transition rates between two given states:

```
───────────────────────────────── R input ─────────────────────────────────

131  #' Return the transition rate for two given states.
132  #'
133  #' @param in_state an integer
134  #' @param out_state an integer
135  #' @param waiting_room an integer (default: 4)
136  #' @param num_barbers an integer  (default: 2)
137  #'
138  #' @return A real
139  get_transition_rate <- function(in_state,
140                                  out_state,
141                                  waiting_room = 4,
142                                  num_barbers = 2){
143    arrival_rate <- 10
144    service_rate <- 4
145
146    capacity <- waiting_room + num_barbers
147    delta <- out_state - in_state
148
149    if (delta == 1) {
150      if (in_state < capacity) {
151        return(arrival_rate)
152      }
153    }
154
155    if (delta == -1) {
156      return(min(in_state, num_barbers) * service_rate)
157    }
158    return(0)
159  }
```

We will not actually use this function but a vectorized version of this:

```
───────────────────────────────── R input ─────────────────────────────────

160  vectorized_get_transition_rate <- Vectorize(
161    get_transition_rate,
162    vectorize.args = c("in_state", "out_state")
163  )
```

This function can now take a vector of inputs for the `in_state` and `out_state` variables which will allow us to simplify the following code that creates the matrices:

R input

```r
#' Return the transition rate matrix Q
#'
#' @param waiting_room an integer (default: 4)
#' @param num_barbers an integer (default: 2)
#'
#' @return A matrix
get_transition_rate_matrix <- function(waiting_room = 4,
                                       num_barbers = 2){
  max_state <- waiting_room + num_barbers

  Q <- outer(0:max_state,
    0:max_state,
    vectorized_get_transition_rate,
    waiting_room = waiting_room,
    num_barbers = num_barbers
  )
  row_sums <- rowSums(Q)

  diag(Q) <- -row_sums
  Q
}
```

Using this we can obtain the matrix Q for our default system:

R input

```r
Q <- get_transition_rate_matrix()
print(Q)
```

which gives:

```
                                    R output
187          [,1] [,2] [,3] [,4] [,5] [,6] [,7]
188    [1,]  -10   10    0    0    0    0    0
189    [2,]    4  -14   10    0    0    0    0
190    [3,]    0    8  -18   10    0    0    0
191    [4,]    0    0    8  -18   10    0    0
192    [5,]    0    0    0    8  -18   10    0
193    [6,]    0    0    0    0    8  -18   10
194    [7,]    0    0    0    0    0    8   -8
```

One immediate thing we can do with this matrix is take the matrix exponential discussed above. To do this, we need to use an R library call expm.

To be able to make use of the nice %>% "pipe" operator we are also going to load the magrittr library. Now if we wanted to see what would happen after .5 time units we obtain:

```
                                    R input
195    library(expm, warn.conflicts = FALSE, quietly = TRUE)
196    library(magrittr, warn.conflicts = FALSE, quietly = TRUE)
197
198    print( (Q * .5) %>% expm %>% round(5))
```

which gives:

```
                                    R output
199            [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]
200    [1,] 0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.08150
201    [2,] 0.08501 0.18292 0.18666 0.17080 0.14377 0.11890 0.11194
202    [3,] 0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346
203    [4,] 0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142
204    [5,] 0.02667 0.07361 0.10005 0.13422 0.17393 0.21890 0.27262
205    [6,] 0.01567 0.04870 0.07552 0.11775 0.17512 0.24484 0.32239
206    [7,] 0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914
```

After 500 time units we obtain:

R input

```
207  print( (Q * 500) %>% expm %>% round(5))
```

---

which gives:

R output

```
208           [,1]     [,2]     [,3]     [,4]     [,5]    [,6]     [,7]
209  [1,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
210  [2,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
211  [3,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
212  [4,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
213  [5,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
214  [6,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
215  [7,]  0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
```

We see that no matter what state (row) we are in, after 500 time units the probabilities are all the same. We could in fact stop our analysis here, however our choice of 500 time units was arbitrary and might not be the correct amount for all possible scenarios, as such we will continue to aim to solve the underlying equation 2.4 directly.

To be able to do this, we will make use of the versatile `pracma` package which includes a number of numerical analysis functions for efficient computations.

```
216  library(pracma, warn.conflicts = FALSE, quietly = TRUE)
217
218  #' Return the steady state vector of any given continuous time
219  #' transition rate matrix
220  #'
221  #' @param Q a transition rate matrix
222  #'
223  #' @return A vector
224  get_steady_state_vector <- function(Q){
225    state_space_size <- dim(Q)[1]
226    A <- rbind(t(Q), 1)
227    b <- c(integer(state_space_size), 1)
228    mldivide(A, b)
229  }
```

This is making use of `pracma`'s `mldivide` function which chooses the best numerical algorithm to find the solution to a given matrix equation $Ax = b$.

So if we now see the steady state vector for our default system:

```
230  print(get_steady_state_vector(Q))
```

we get:

```
231             [,1]
232  [1,]  0.03430888
233  [2,]  0.08577220
234  [3,]  0.10721525
235  [4,]  0.13401906
236  [5,]  0.16752383
237  [6,]  0.20940479
238  [7,]  0.26175598
```

We can see that the shop is expected to be empty approximately 3.4% of the time and full 26.2% of the time.

The final piece of this puzzle is to create a single function that uses all of the above to just return the probability of the shop being full.

```
R input

239  #' Return the probability of the barber shop being full
240  #'
241  #' @param waiting_room (default: 4)
242  #' @param num_barbers (default: 2)
243  #'
244  #' @return A real
245  get_probability_of_full_shop <- function(waiting_room = 4,
246                                           num_barbers = 2){
247    arrival_rate <- 10
248    service_rate <- 4
249    pi <- get_transition_rate_matrix(
250      waiting_room = waiting_room,
251      num_barbers = num_barbers
252      ) %>%
253      get_steady_state_vector()
254
255    capacity <- waiting_room + num_barbers
256    pi[capacity + 1]
257  }
```

Now we can run this code efficiently with both scenarios:

```
R input

258  print(get_probability_of_full_shop(waiting_room = 6))
```

which decreases the probability of a full shop to:

```
R output

259  [1] 0.2355699
```

but adding another barber and chair:

```
──────────────────────── R input ────────────────────────

260   print(get_probability_of_full_shop(num_barbers = 3))
```

gives:

```
──────────────────────── R output ────────────────────────

261   [1] 0.0786359
```

2.5   RESEARCH

TBA

Discrete Event Simulation

C OMPLEX situations further compounded by randomness appear throughout our daily lives. For example, data flowing through a computer network, patients being treated at an emergency services, and daily commutes to work. Mathematics can be used to understand these complex situations so as to make predictions which in turn can be used to make improvements. One tool used to do this is to let a computer create a dynamic virtual representation of the scenario in question, the particular type we are going to cover here is called Discrete Event Simulation.

3.1   PROBLEM

Consider the following situation: a bicycle repair shop would like reconfigure their set-up in order to guarantee that all bicycles processed by the repair shop take a maximum of 30 minutes. Their current set-up is as follows:

- Bicycles arrive randomly at the shop at a rate of 15 per hour.

- They wait in line to be seen at an inspection counter, manned by one member of staff who can inspect one bicycle at a time. On average an inspection takes around 3 minutes.

- After inspection it is found that around 20% of bicycles do not need repair, and they are then ready for collection.

- After inspection is is found that around 80% of bicycles go on to be repaired. These then wait in line outside the repair workshop, which is manned by two members of staff who can each repair one bicycle at a time. On average a repair takes around 6 minutes.

- After repair the bicycles are ready for collection.

A diagram of the system is shown in Figure 3.1

We can also assume that there is infinite capacity at the bicycle repair shop for waiting bicycles. The shop will hire and extra member of staff in order to meet their target of a maximum time in the system of 30 minutes. They would like to know if they should work on the inspection counter or in the repair workshop?
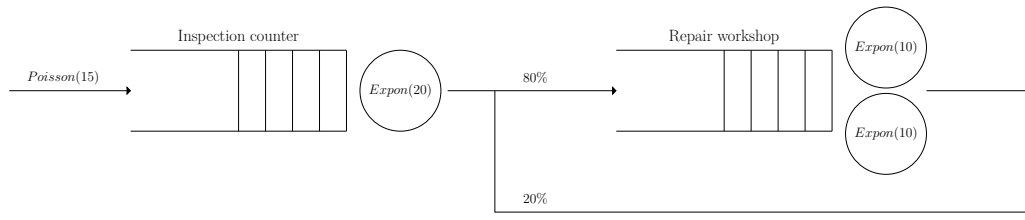
Figure 3.1   Diagrammatic representation of the bicycle repair shop as a queuing system.

3.2   THEORY

A number of the events that govern the behaviour of the bicycle shop above are probabilistic. For example the times that bicycles arrive at the shop, the duration of the inspection and repairs, and whether the bicycle would need to go on to be repaired or not. When a number of these probabilistic events are arranged in a complex system such as the bicycle shop, using analytical methods to manipulate these probabilities can become difficult. One method to deal with this is *simulation.*

Consider one probabilistic event, rolling a die. A die has six sides numbered 1 to 6, each side is equally likely to land. Therefore the probability of rolling a 1 is $\frac{1}{6}$, the probability of rolling a 2 is $\frac{1}{6}$, and so on. This means that that if we roll the die a large number of times, we would except $\frac{1}{6}$ of those rolls to be a 1. This is called the *law of large numbers.*

Now imagine we have an event in which we do not know the analytical probability of it occurring. Consider rolling a weighted die, in this case a die in which the probability of obtaining one number is much greater than the others. How can we estimate the probability of obtaining a 5 on this die?

Rolling the weighted die once does not give us much information. However due to the law of large numbers, we can roll this die a number of times, and find the proportion of those rolls which gave a 5. The more times we roll the die, the closer this proportion approaches the underlying probability of obtaining a 5.

For a complex system such as the bicycle shop, we would like to estimate the proportion of bicycles that take longer than 30 minutes to be processed. As it is a complex system it is difficult to work this out analytically. So, just like the weighted die, we would like to observe this system a number of times and record the overall proportions of bicycles spending longer than 30 minutes in the shop, which will converge to the true underlying proportion. However unlike rolling a weighted die, it it costly to observe this shop over a number of days with identical conditions. In this case it is costly in terms of time, as the repair shop already exists. However some scenarios, for example the scenario where the repair shop hires and additional member of staff, do not yet exist, so observing this this would be costly in terms of money also. We can however build a virtual representation of this complex system on a computer, and observe a virtual day of work much more quickly and much less costly on the computer, similar to a video game.

In order to do this, the computer needs to be able to generate random outcomes of

each of the smaller events that make up the large complex system. Generating random events are essentially doing things to random numbers, that need to be generated.

Computers are deterministic, therefore true randomness is not always possible. They can however generate pseudorandom numbers: sequences of numbers that look like random numbers, but are entirely determined from the previous numbers in the sequence. Most programming languages have methods of doing this.

In order to simulate an event we can again manipulate the law of large numbers. Let $X \sim U(0, 1)$, a uniformly pseudorandom variable between 0 and 1. Let D be the outcome of a roll of an unbiased die. Then D can be defined as:

$$D = \begin{cases} 1 & \text{if } 0 \leq X < \frac{1}{6} \\ 2 & \text{if } \frac{1}{6} \leq X < \frac{2}{6} \\ 3 & \text{if } \frac{2}{6} \leq X < \frac{3}{6} \\ 4 & \text{if } \frac{3}{6} \leq X < \frac{4}{6} \\ 5 & \text{if } \frac{4}{6} \leq X < \frac{5}{6} \\ 6 & \text{if } \frac{5}{6} \leq X < 1 \end{cases} \tag{3.1}$$

The bicycle repair shop is a system made up of interactions of a number of other simpler random events. This can be thought of as many interactions of random variables, each generated using pseudorandom numbers.

In this case the fundamental random events that need to be generated are:

- the time each bicycle arrives to the repair shop,

- the time each bicycle spends at the inspection counter,

- whether each bicycle needs to go on the the repair workshop,

- the time each those bicycles spends at the repair shop.

As the simulation progresses these events will be generated, and will interact together as described in Section 4.1. The proportion of customers spending longer than 30 minutes in the shop can then be counted. This proportion itself is a random variable, and so just like the weighted die, running this simulation once does not give us much information. But we can run the simulation many times and take an average proportion, to smooth out any variability.

The process outlined above is a particular implementation of Monte Carlo simulation called *discrete event simulation*, which generates pseudorandom numbers and observes their interactions. In practice there are two main approaches to simulating complex probabilistic systems such as this one: the *event scheduling* approach, and *process based* simulation. It just so happens that the main implementations in Python and R use each of these approaches, so you will see both approaches used here.

3.2.1 Event Scheduling Approach

When using the event scheduling approach, we can think of the 'virtual representation' of the system as being the facilities that the bicycles use, shown in Figure 3.1.

Then we let entities (the bicycles) interact with these facilities. It is these facilities that determine how the entities behave.

In a simulation that uses an event scheduling approach, a key concept is that events occur that cause further events to occur in the future, either immediately or after a delay, such as after some time in service. In the bicycle shop examples of such events include a bicycle joining a queue, a bicycle beginning service, and a bicycle finishing service. At each event the event list is updated, and the clock then jumps forward to the next event in this updated list.

3.2.2 Process Based Simulation

When using process based simulation, we can think of the 'virtual representation' of the system as being the sequence of actions that each entity (the bicycles) must take, and these sequences of actions might contain delays as a number of entities seize and release a finite amount of resources. It is the sequence of actions that determine how the entities behave.

For the bicycle repair shop an example of one possible sequence of actions would be:

arrive → seize inspection counter → delay → release inspection counter → seize repair shop → delay → release repair shop → leave

The scheduled delays in this sequence of events correspond to the time spend being inspected and the time spend being repaired. Waiting in line for service at these facilities are not included in the sequence of events; that is implicit by the 'seize' and 'release' actions, as an entity will wait for a free resource before seizing one. Therefore in process based simulations, in addition to defining a sequence of events, resource types and their numbers also need to be defined.

3.3 SOLVING WITH PYTHON

In this book we will use the Ciw library in order to conduct discrete event simulation in Python. Ciw uses the event scheduling approach, which means we must define the system's facilities, and then let customers loose to interact with them.

In this case there are two facilities to define: the inspection desk and the repair workshop. Let's order these as so. For each of these we need to define:

- the distribution of times between consecutive bicycles arriving,

- the distribution of times the bicycles spend in service,

- the number of servers available,

- the probability of routing to each of the other facilities after service.

In this case we will assume that the time between consecutive arrivals follow a exponential distribution, and that the service times also follow exponential distributions. These are common assumptions for this sort of queueing system.

In Ciw, these are defined in a Network object, created using the `ciw.create_network`

function. The function below creates a Network object that defines the for a given set of parameters bicycle repair shop:

```python
import ciw


def build_network_object(
    num_inspectors=1, num_repairers=2,
):
    """Returns a Network object that defines the repair shop.

    Args:
        num_inspectors: a positive integer (default: 1)
        num_repairers: a positive integer (default: 2)

    Returns:
        a Ciw network object
    """
    arrival_rate = 15
    inspection_rate = 20
    repair_rate = 10
    prob_need_repair = 0.8
    N = ciw.create_network(
        arrival_distributions=[
            ciw.dists.Exponential(arrival_rate),
            ciw.dists.NoArrivals(),
        ],
        service_distributions=[
            ciw.dists.Exponential(inspection_rate),
            ciw.dists.Exponential(repair_rate),
        ],
        number_of_servers=[num_inspectors, num_repairers],
        routing=[[0.0, prob_need_repair], [0.0, 0.0]],
    )
    return N
```

A Network object is used by Ciw to access system parameters. For example one piece of information it holds is the number of nodes of the system:

```
                        ┌─────────── Python input ───────────┐
294   N = build_network_object()
295   print(N.number_of_nodes)
```

which gives:

```
                        ┌─────────── Python output ───────────┐
296   2
```

Now we have defined the system, we need to use this to build the virtual representation of the system: in Ciw this is a Simulation object. Once this is built the simulation can be run, that is observe it for one virtual day. The following function does this:

```
                        ┌─────────── Python input ───────────┐
297   def run_simulation(network, seed=0):
298       """Builds a simulation object and runs it for 8 time units.
299
300       Args:
301           network: a Ciw network object
302           seed: a float (default: 0)
303
304       Returns:
305           a Ciw simulation object after a run of the simulation
306       """
307       max_time = 8
308       ciw.seed(seed)
309       Q = ciw.Simulation(network)
310       Q.simulate_until_max_time(max_time)
311       return Q
```

Notice here a random seed is set. This is because there is some element of randomness when initialising this object, and much randomness in running the simulation, and in order to ensure reproducible results we force the pseudorandom number generator to produce the same sequence of pseudorandom numbers each time. Notice also that the simulation always begins with an empty system, so the first bicycle to arrive will never wait for service. Depending on the situation this may be an unwanted

feature, though not in this case as it is reasonable to assume that the bicycle shop will begin the day with no customers.

Now we wish to count the number of bicycles that have finished service, and to count the number of those whose entire journey through the system lasted longer than 0.5 hours. In order to do so, we'll use the `pandas` library for efficient manipulation of data frames.

```python
import pandas as pd


def get_proportion(Q):
    """Returns the proportion of bicycles spending over a given
    limit at the repair shop.

    Args:
        Q: a Ciw simulation object after a run of the
            simulation

    Returns:
        a real
    """
    limit = 0.5
    inds = Q.nodes[-1].all_individuals
    recs = pd.DataFrame(
        dr for ind in inds for dr in ind.data_records
    )
    recs["total_time"] = (
        recs["exit_date"] - recs["arrival_date"]
    )
    total_times = recs.groupby("id_number")["total_time"].sum()
    return (total_times > limit).mean()
```

Altogether these functions can define the system, run one day of our system, and then find the proportion of bicycles spending over half an hour in the shop:

```
                           ── Python input ──
336    N = build_network_object()
337    Q = run_simulation(N)
338    p = get_proportion(Q)
339    print(round(p, 6))
```

This piece of code gives

```
                           ── Python output ──
340    0.261261
```

meaning 26.13% of all bicycles spent longer than half an hour at the repair shop.

However this particular day may have contained a number of extreme events. For a more accurate proportion this experiment should be repeated, and an average proportion taken. In order to do so, let's write a function that performs the above experiment over a number of trials, then finds an average proportion:

```
                          Python input

341   def get_average_proportion(num_inspectors=1, num_repairers=2):
342       """Returns the average proportion of bicycles spending over
343       a given limit at the repair shop.
344
345       Args:
346           num_inspectors: a positive integer (default: 1)
347           num_repairers: a positive integer (default: 2)
348
349       Returns:
350           a real
351       """
352       num_trials = 100
353       N = build_network_object(
354           num_inspectors=num_inspectors,
355           num_repairers=num_repairers,
356       )
357       proportions = []
358       for trial in range(num_trials):
359           Q = run_simulation(N, seed=trial)
360           proportion = get_proportion(Q=Q)
361           proportions.append(proportion)
362       return sum(proportions) / num_trials
```

This can be used to find the average proportion over 100 trials for the current system of one inspector and two repair people:

```
                          Python input

363   p = get_average_proportion(num_inspectors=1, num_repairers=2)
364   print(round(p, 6))
```

which gives:

```
                          Python output

365   0.159354
```

that is, on average 15.94% of bicycles will spend longer than 30 minutes at the repair shop.

Now consider the two possible future scenarios we wish top compare: hiring an extra member of staff to serve at the inspection desk, or hiring an extra member of staff at the repair workshop. Which scenario yields a smaller proportion of bicycles spending over 30 minutes at the shop? Let's investigate. First look at the situation where the additional member of staff works at the inspection desk:

_____ Python input _____

```
366  p = get_average_proportion(num_inspectors=2, num_repairers=2)
367  print(round(p, 6))
```

which gives:

_____ Python output _____

```
368  0.038477
```

that is 3.85% of bicycles.

Now look at the situation where the additional member of staff works at the repair workshop:

_____ Python input _____

```
369  p = get_average_proportion(num_inspectors=1, num_repairers=3)
370  print(round(p, 6))
```

which gives:

_____ Python output _____

```
371  0.103591
```

that is 10.36% of bicycles.

Therefore an additional member of staff at the inspection desk would be more beneficial than an additional member of staff at the repair workshop.
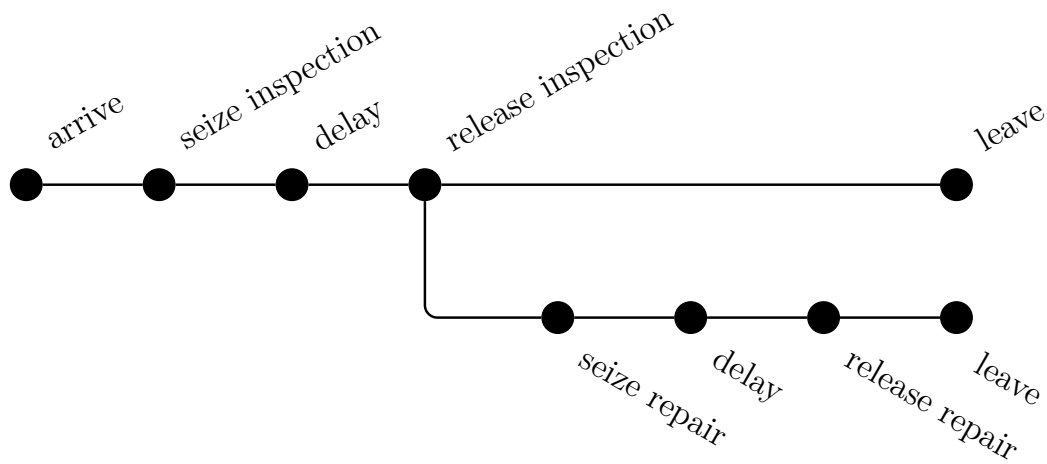
Figure 3.2 Diagrammatic representation of the forked trajectories a bicycle can take

3.4 SOLVING WITH R

In this book we will use the Simmer package in order to conduct discrete event simulation in R. Simmer uses the process based approach, which means we must define the each bicycle's sequence of actions, and then generate a number of bicycles with these sequences.

In Simmer these sequences of actions are made up of trajectories. The diagram in Figure 3.2 shows the branched trajectories than a bicycle would take at the repair shop:

The function below defines a simmer object that describes these trajectories:

R input

```
372  library(simmer)
373
374  #' Returns a simmer trajectory object outlining the bicycles
375  #' path through the repair shop
376  #'
377  #' @return A simmer trajectory object
378  define_bicycle_trajectories <- function() {
379    inspection_rate <- 20
380    repair_rate <- 10
381    prob_need_repair <- 0.8
382    bicycle <-
383      trajectory("Inspection") %>%
384      seize("Inspector") %>%
385      timeout(function() {
386        rexp(1, inspection_rate)
387      }) %>%
388      release("Inspector") %>%
389      branch(
390        function() (runif(1) < prob_need_repair),
391        continue = c(F),
392        trajectory("Repair") %>%
393          seize("Repairer") %>%
394          timeout(function() {
395            rexp(1, repair_rate)
396          }) %>%
397          release("Repairer"),
398        trajectory("Out")
399      )
400    return(bicycle)
401  }
```

These trajectories are not very useful alone, we are yet to define the resources used, or a way to generate bicycles with these trajectories. This is done in the function below, which begins by defining a `repair_shop` with one resource labelled "Inspector", and two resources labelled "Repairer". Once this is built the simulation can be run, that is observe it for one virtual day. The following function does all this:

```
     ┌─────────────── R input ───────────────┐

402  #' Runs one trial of the simulation.
403  #'
404  #' @param bicycle a simmer trajectory object
405  #' @param num_inspectors positive integer (default: 1)
406  #' @param num_repairers positive integer (default: 2)
407  #' @param seed a float (default: 0)
408  #'
409  #' @return A simmer simulation object after one run of
410  #;          the simulation
411  run_simulation <- function(bicycle,
412                             num_inspectors = 1,
413                             num_repairers = 2,
414                             seed = 0) {
415    arrival_rate <- 15
416    max_time <- 8
417    repair_shop <-
418      simmer("Repair Shop") %>%
419      add_resource("Inspector", num_inspectors) %>%
420      add_resource("Repairer", num_repairers) %>%
421      add_generator("Bicycle", bicycle, function() {
422        rexp(1, arrival_rate)
423      })
424
425    set.seed(seed)
426    repair_shop %>% run(until = 8)
427    return(repair_shop)
428  }
```

Notice here a random seed is set. This is because there are elements of randomness when running the simulation, and in order to ensure reproducible results we force the pseudorandom number generator to produce the same sequence of pseudorandom numbers each time. Notice also that the simulation always begins with an empty system, so the first bicycle to arrive will never wait for service. Depending on the situation this may be an unwanted feature, though not in this case as it is reasonable to assume that the bicycle shop will begin the day with no customers.

Now we wish to count the number of bicycles that have finished service, and to count the number of those whose entire journey through the system lasted longer than 0.5 hours. Using Simmer's `get_mon_arrivals`() function we can get a data frame of records to manipulate.

R input

```r
#' Returns the proportion of bicycles spending over 30
#' minutes in the repair shop
#'
#' @param repair_shop a simmer simulation object
#'
#' @return a float between 0 and 1
get_proportion <- function(repair_shop) {
  limit <- 0.5
  recs <- repair_shop %>% get_mon_arrivals()
  total_times <- recs$end_time - recs$start_time
  return(mean(total_times > 0.5))
}
```

Altogether these functions can define the system, run one day of our system, and then find the proportion of bicycles spending over half an hour in the shop:

R input

```r
bicycle <- define_bicycle_trajectories()
repair_shop <- run_simulation(bicycle = bicycle)
print(get_proportion(repair_shop = repair_shop))
```

This piece of code gives

R output

```
[1] 0.04032258
```

meaning 4.03% of all bicycles spent longer than half an hour at the repair shop.

However this particular day may have contained a number of extreme events. For a more accurate proportion this experiment should be repeated, and an average proportion taken. In order to do so, let's write a function that performs the above experiment over a number of trials, then finds an average proportion:

```
           R input

445  #' Returns the average proportion of bicycles spending over
446  #' a given limit at the repair shop.
447  #'
448  #' @param num_inspectors positive integer (default: 1)
449  #' @param num_repairers positive integer (default: 2)
450
451  #' @return a float between 0 and 1
452  get_average_proportion <- function(num_inspectors = 1,
453                                     num_repairers = 2) {
454    num_trials <- 100
455    bicycle <- define_bicycle_trajectories()
456    proportions <- c()
457    for (trial in 1:num_trials) {
458      repair_shop <- run_simulation(
459        bicycle = bicycle,
460        num_inspectors = num_inspectors,
461        num_repairers = num_repairers,
462        seed = trial
463      )
464      proportion <- get_proportion(
465        repair_shop = repair_shop
466      )
467      proportions[trial] <- proportion
468    }
469    return(mean(proportions))
470  }
```

This can be used to find the average proportion over 100 trials:

```
           R input

471  print(
472    get_average_proportion(
473      num_inspectors = 1,
474      num_repairers = 2)
475  )
```

which gives:

```
R output
```

476   `[1] 0.1551579`

that is, on average 15.52% of bicycles will spend longer than 30 minutes at the repair shop.

Now consider the two possible future scenarios we wish top compare: hiring an extra member of staff to serve at the inspection desk, or hiring an extra member of staff at the repair workshop. Which scenario yields a smaller proportion of bicycles spending over 30 minutes at the shop? Let's investigate. First look at the situation where the additional member of staff works at the inspection desk:

```
R input
```

```
477    print(
478      get_average_proportion(
479        num_inspectors = 2,
480        num_repairers = 2)
481    )
```

which gives:

```
R output
```

482   `[1] 0.04115338`

that is 4.12% of bicycles.

Now look at the situation where the additional member of staff works at the repair workshop:

```
R input
```

```
483    print(
484      get_average_proportion(
485        num_inspectors = 1,
486        num_repairers = 3)
487    )
```

which gives:

```
┌─────────────────────── R output ───────────────────────┐
│                                                         │
│  ┌───────────────────────────────────────────────────┐ │
│  │ [1] 0.1000899                                     │ │
│  └───────────────────────────────────────────────────┘ │
│                                                         │
└─────────────────────────────────────────────────────────┘
```

that is 10.01% of bicycles.

Therefore an additional member of staff at the inspection desk would be more beneficial than an additional member of staff at the repair workshop.

3.5   RESEARCH

TBA

III

Dynamical Systems

Modelling with Differential Equations

S YSTEMS often change in a way that depends on their current state. For example, the speed at which a cup of coffee cools down depends on its current temperature. These types of systems are called dynamical systems and are modelled mathematically using differential equations. In this chapter we will consider a direct solution approach using symbolic mathematics.

4.1 PROBLEM

Consider the following situation: the entire population of a small rural town has caught a cold. All 100 individuals will recover at an average rate of 2 per day. The town leadership have noticed that being ill costs approximately £10 per day, this is due to general lack of productivity, poorer mood and other intangible aspects. They need to decide whether or not to order cold medicine which would **double** the recover rate. The cost of of the cold medicine is a one off cost of £5 per person.

4.2 THEORY

In the case of this town, the overall rate at which people get better is dependent on the number of people in how are ill. This can be represented mathematically using a differential equation which is a way of relating the rate of change of a system to the state of the system itself.

In general if we are interested in some variable x over time t the differential function equation will be of the form:

$$\frac{dx}{dt} = f(x) \tag{4.1}$$

For some function f. In our case, if we denote the number of infected individuals as I where we implicitly mean that I is a function of time: $I = I(t)$ and the rate at which individuals recover by α then the differential equation that describes the above situation is:

$$\frac{dI}{dt} = -\alpha I \tag{4.2}$$

Finding a solution to this differential equation means finding an expression for I that when differentiated gives $-alphaI$.

In this particular case, one such function is:

$$I(t) = e^{-\alpha t} \tag{4.3}$$

However, $I(0) = 1$ whereas for our problem we know that at time $t = 0$ there are 100 infected individuals. Indeed a differential equation defines a family of solutions and we need to know some sort of initial (also referred to as boundary) condition to have the exact solution. Which in this case would be:

$$I(t) = 100e^{-\alpha t} \tag{4.4}$$

To evaluate the cost we then need to know the sum of the values of that function over time. Integration gives us exactly this, so the cost would be:

$$K \int_0^\infty I(t)dt \tag{4.5}$$

where K is the cost per person per unit time.

In the upcoming sections we will confirm and use code to carry out the above efficiently so as to answer the original question.

4.3   SOLVING WITH PYTHON

The first step we will take is to write a function to obtain the differential equation. Note that here we will be using the Python library `sympy` which allows us to carry out symbolic calculations.

```
                              Python input

489    import sympy as sym

490

491    t = sym.Symbol("t")
492    alpha = sym.Symbol("alpha")
493    I_0 = sym.Symbol("I_0")
494    I = sym.Function("I")

495

496

497    def get_equation(alpha=alpha):
498        """Return the differential equation.

499

500        Args:
501            alpha: a float (default: symbolic alpha)

502

503        Returns:
504            A symbolic equation
505        """
506        return sym.Eq(sym.Derivative(I(t), t), -alpha * I(t))
```

Using this we can get the equation that defines the population change over time:

```
                              Python input

507    eq = get_equation()
508    print(eq)
```

which gives:

```
                              Python output

509    Eq(Derivative(I(t), t), -alpha*I(t))
```

Note that if you are using Jupyter then your output will actually be a well rendered mathematical equation:

$$\frac{d}{dt}I(t) = -\alpha I(t)$$

Note that we can pass a value to α if we want to:

_____ Python input _____

```
510  eq = get_equation(alpha=1)
511  print(eq)
```

_____ Python output _____

```
512  Eq(Derivative(I(t), t), -I(t))
```

We will now write a function to obtain the solution to this differential

_____ Python input _____

```
513  def get_solution(I_0=I_0, alpha=alpha):
514      """Return the solution to the differential equation.
515
516      Args:
517          I_0: a float (default: symbolic I_0)
518          alpha: a float (default: symbolic alpha)
519
520      Returns:
521          A symbolic equation
522      """
523      eq = get_equation(alpha=alpha)
524      return sym.dsolve(eq, I(t), ics={I(0): I_0})
```

We can verify the solution discussed previously:

_____ Python input _____

```
525  sol = get_solution()
526  print(sol)
```

which gives:

```
527   Eq(I(t), I_0*exp(-alpha*t))
```

$$I(t) = I_0 e^{-\alpha t}$$

We can use sympy itself to verify the result, by taking the derivative of the right hand side of our solution.

```
528   print(sym.diff(sol.rhs, t) == -alpha * sol.rhs)
```

which gives:

```
529   True
```

All of the above has given us the general solution in terms of $I(0) = I_0$ and α however we have written the code in such a way as we can pass the actual parameters:

```
530   sol = get_solution(alpha=2, I_0=100)
531   print(sol)
```

which gives:

```
532   Eq(I(t), 100*exp(-2*t))
```

Now, to calculate the cost we will write a function to integrate our result:

_____ Python input _____

```
533  def get_cost(
534      I_0=I_0, alpha=alpha, cost_per_person=10, cost_of_cure=0,
535  ):
536      """Return the cost.
537
538      Args:
539          I_0: a float (default: symbolic I_0)
540          alpha: a float (default: symbolic alpha)
541          cost_per_person: a float (default: 10)
542          cost_of_cure: a float (default: 0)
543
544      Returns:
545          A symbolic expression
546      """
547      I_sol = get_solution(I_0=I_0, alpha=alpha)
548      return (
549          sym.integrate(I_sol.rhs, (t, 0, sym.oo))
550          * cost_per_person
551          + cost_of_cure * I_0
552      )
```

We can now obtain the cost without purchasing the cure:

_____ Python input _____

```
553  I_0 = 100
554  alpha = 2
555  cost_without_cure = get_cost(I_0=I_0, alpha=alpha)
556  print(cost_without_cure)
```

which gives:

_____ Python output _____

```
557  500
```

The cost with cure can use the above with a modified α and a non zero cost of the cure itself:

```
                              ─── Python input ───

558   cost_of_cure = 5
559   cost_with_cure = get_cost(
560       I_0=I_0, alpha=2 * alpha, cost_of_cure=cost_of_cure
561   )
562   print(cost_with_cure)
```

which gives:

```
                              ─── Python output ───

563   750
```

So given the current parameters it is not worth purchasing the cure.

4.4   SOLVING WITH R

R does not have a bespoke symbolic mathematics library however it has an interface
to use Python's sympy called rSymPy. In general we would recommend to use Python
directly when needing to carry out symbolic mathematical calculations however this
is not always possible. Furthermore, the development of rSymPy seems to no longer
be active and due to the dependency on Java could be problematic to use. One of
other the difficulties of using rSymPy is that it works by passing string back and forth
to Python's sympy. As such, we will need to have more of a manual approach and so
we will not be able to write the modular code we have written throughout the book.
    First let us setup and solve the differential equation:

```
                              ─── R input ───

564   library("rSymPy", warn.conflicts = FALSE, quietly = TRUE)
565
566   t <- Var("t")
567   alpha <- Var("alpha")
568   I <- Var("I")
569   sympy("dsolve(Derivative(I(t), t) - alpha *I(t), I(t))")
```

this will give us:

─── R output ───

```
570    [1] "C1*exp(alpha*t)"
```

We can then solve the equation directly to find the initial condition:

─── R input ───

```
571    sympy("solve(C1*exp(alpha*0) - I_0, C1)")
```

This gives:

─── R output ───

```
572    [1] "[I_0]"
```

Note here that this is in fact using a much older version of Sympy so we cannot take advantage of some of the newer commands.

So we now the form of our particular solution, thus we can now write a function that will give us the cost:

R input

```
573   #' Return the cost
574   #'
575   #' @param I_0 a float (default: symbolic I_0)
576   #' @param alpha a float (default: symbolic alpha)
577   #' @param cost_per_person a float (default: 10)
578   #' @param cost_of_cure a float (default: 0)
579   #'
580   #' @return A symbolic expression
581   get_cost <- function(I_0 = Var("I_0"),
582                        alpha = Var("alpha"),
583                        cost_per_person = 10,
584                        cost_of_cure = 0) {
585     t <- Var("t")
586     solution_cmd <- sprintf(
587       "I_sol = %s * exp(-%s * t)",
588       I_0,
589       alpha
590     )
591     sympy(solution_cmd)
592     cost_cmd <- sprintf(
593       "integrate(I_sol, (t, 0, oo)) * %s + %s * %s",
594       cost_per_person, cost_of_cure,
595       I_0
596     )
597     return(sympy(cost_cmd))
598   }
```

Using this:

R input

```
599   I_0 <- 100
600   alpha <- 2
601   cost_without_cure <- get_cost(I_0 = I_0, alpha = alpha)
602   print(cost_without_cure)
```

which gives:

R output

```
500
```

The cost with cure can use the above with a modified α and a non zero cost of the cure itself:

R input

```
cost_of_cure <- 5
cost_with_cure <- get_cost(
    I_0 = I_0, alpha = 2 * alpha, cost_of_cure = cost_of_cure
)
print(cost_with_cure)
```
---

which gives:

R output

```
750
```

So given the current parameters it is not worth purchasing the cure.

4.5 RESEARCH

TBA

Bibliography