
Half Title Page



Title Page

LOC Page

Vince: to Riggins
Geraint: also, to Riggins



Contents

Foreword	xi
Preface	xiii
Contributors	xv
SECTION I Getting Started	
CHAPTER 1 ■ Introduction	3
1.1 WHO IS THIS BOOK FOR?	3
1.2 WHAT DO WE MEAN BY APPLIED MATHEMATICS?	3
1.3 WHAT IS OPEN SOURCE SOFTWARE	4
1.4 HOW TO GET THE MOST OUT OF THIS BOOK	4
SECTION II Probabilistic Modelling	
CHAPTER 2 ■ Markov Chains	9
2.1 PROBLEM	9
2.2 THEORY	9
2.3 SOLVING WITH PYTHON	11
2.4 SOLVING WITH R	18
2.5 RESEARCH	25
CHAPTER 3 ■ Discrete Event Simulation	27
3.1 TYPICAL PROBLEM	27
3.2 THEORY	28
3.2.1 Event Scheduling Approach	30
3.2.2 Process Based Simulation	30
3.3 SOLVING WITH PYTHON	30

3.4	SOLVING WITH R	37
3.5	RESEARCH HIGHLIGHTS	43

SECTION III Dynamical Systems

CHAPTER	4 ■ Modelling with Differential Equations	47
---------	---	----

4.1	PROBLEM	47
4.2	THEORY	47
4.3	SOLVING WITH PYTHON	48
4.4	SOLVING WITH R	53
4.5	RESEARCH	57

CHAPTER	5 ■ Systems dynamics	59
---------	----------------------	----

5.1	PROBLEM	59
5.2	THEORY	59
5.3	SOLVING WITH PYTHON	62
5.4	SOLVING WITH R	70
5.5	RESEARCH	77

SECTION IV Emergent Behaviour

CHAPTER	6 ■ Game Theory	81
---------	-----------------	----

6.1	PROBLEM	81
6.2	THEORY	82
6.3	SOLVING WITH PYTHON	83
6.4	SOLVING WITH R	87
6.5	RESEARCH	90

CHAPTER	7 ■ Agent Based Simulation	91
---------	----------------------------	----

7.1	PROBLEM	91
7.2	THEORY	91
7.3	SOLVING WITH PYTHON	94
7.4	SOLVING WITH R	99
7.5	RESEARCH	105

SECTION V Optimisation

CHAPTER	8 ■ Linear programming	109
8.1	PROBLEM	109
8.2	THEORY	110
8.3	SOLVING WITH PYTHON	114
8.4	SOLVING WITH R	118
8.5	RESEARCH	126
CHAPTER	9 ■ Heuristics	127
9.1	PROBLEM	127
9.2	THEORY	127
9.3	SOLVING WITH PYTHON	130
9.4	SOLVING WITH R	138
9.5	RESEARCH	146
	Bibliography	149



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

Greg 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

xvi ■ Contributors

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

THANK 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

MANY real world situations have some level of unpredictability through randomness: 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 can be expected 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 here 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: the number of customers present. If that number is 1 this implies that

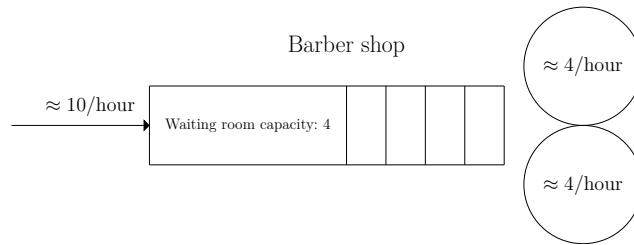


Figure 2.1 Diagrammatic representation of the barber shop as a queuing system.

1 customer is currently having their hair cut. If that number is 5 this implies that 2 customers are being served and 3 are waiting. The entire set of values that this value can take is a finite set of integers from 0 to 6, this set, in general, is called the *state space*. If the system is full (all barbers and waiting room occupied) then the Markov chain is in state 6 and if there is no one at the shop then it is in state 0. This is denoted mathematically as:

$$S = \{0, 1, 2, 3, 4, 5, 6\} \quad (2.1)$$

The state increases when people arrive and this happens at a rate of change of 10. The state decreases when people are served and this happens at a rate of 4 per active server. In both cases it is assumed that no 2 events can occur at the same time.

The rules that govern how to move between these states can be defined in 2 ways:

- Using probabilities of changing state (or not) in a well defined time interval. This is called a discrete Markov chain.
- Using rates of change from one state to another. This is called a continuous time Markov chain.

The barber shop will be considered as a continuous Markov chain as shown in Figure 2.2

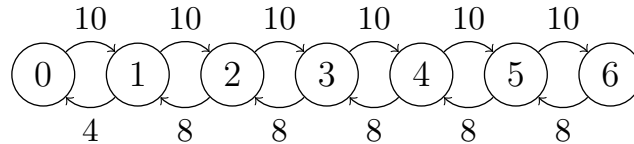


Figure 2.2 Diagrammatic representation of the state space and the transition rates

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 the probability of a customer finishing service within the next 5 minutes does not change if they have been having their hair cut for 3 minutes already.

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:

$$Q = \begin{pmatrix} -10 & 10 & 0 & 0 & 0 & 0 & 0 \\ 4 & -14 & 10 & 0 & 0 & 0 & 0 \\ 0 & 8 & -18 & 10 & 0 & 0 & 0 \\ 0 & 0 & 8 & -18 & 10 & 0 & 0 \\ 0 & 0 & 0 & 8 & -18 & 10 & 0 \\ 0 & 0 & 0 & 0 & 8 & -18 & 10 \\ 0 & 0 & 0 & 0 & 0 & 8 & -8 \end{pmatrix} \quad (2.2)$$

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 .

The matrix Q can be used to understand the probability of being in a given state after t time units. This 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 . Using a mathematical tool called the matrix exponential the value of P_t can be calculated numerically.

$$P_t = e^{Qt} \quad (2.3)$$

What is also useful is understanding the long run behaviour of the system. This allows us to answer questions such as “what state is the system 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:

$$\pi Q = 0 \quad (2.4)$$

with the following constraint:

$$\sum_{i=1}^n \pi_i = 1 \quad (2.5)$$

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

2.3 SOLVING WITH PYTHON

The first step is to write a function to obtain the transition rates between 2 given states:

Python input

```

1 def get_transition_rate(
2     in_state,
3     out_state,
4     waiting_room=4,
5     num_barbers=2,
6 ):
7     """Return the transition rate for 2 given states.
8
9     Args:
10         in_state: an integer
11         out_state: an integer
12         waiting_room: an integer (default: 4)
13         num_barbers: an integer (default: 2)
14
15     Returns:
16         A real.
17     """
18     arrival_rate = 10
19     service_rate = 4
20
21     capacity = waiting_room + num_barbers
22     delta = out_state - in_state
23
24     if delta == 1 and in_state < capacity:
25         return arrival_rate
26
27     if delta == -1:
28         return min(in_state, num_barbers) * service_rate
29
30     return 0

```

Next, a function that creates an entire transition rate matrix Q for a given problem is written. The `numpy` library will be used to handle all the linear algebra and the `itertools` library for some iterations:

Python input

```

31 import itertools
32 import numpy as np
33
34
35 def get_transition_rate_matrix(waiting_room=4, num_barbers=2):
36     """Return the transition matrix Q.
37
38     Args:
39         waiting_room: an integer (default: 4)
40         num_barbers: an integer (default: 2)
41
42     Returns:
43         A matrix.
44     """
45     capacity = waiting_room + num_barbers
46     state_pairs = itertools.product(
47         range(capacity + 1), repeat=2
48     )
49
50     flat_transition_rates = [
51         get_transition_rate(
52             in_state=in_state,
53             out_state=out_state,
54             waiting_room=waiting_room,
55             num_barbers=num_barbers,
56         )
57         for in_state, out_state in state_pairs
58     ]
59     transition_rates = np.reshape(
60         flat_transition_rates, (capacity + 1, capacity + 1)
61     )
62     np.fill_diagonal(
63         transition_rates, -transition_rates.sum(axis=1)
64     )
65
66     return transition_rates

```

Using this the matrix Q for the default system can be obtained:

Python input

```

67 Q = get_transition_rate_matrix()
68 print(Q)

```

which gives:

Python output

```

69 [[-10  10  0  0  0  0  0]
70 [  4 -14 10  0  0  0  0]
71 [  0  8 -18 10  0  0  0]
72 [  0  0  8 -18 10  0  0]
73 [  0  0  0  8 -18 10  0]
74 [  0  0  0  0  8 -18 10]
75 [  0  0  0  0  0  8 -8]]

```

Here, the matrix exponential will be used as discussed above, using the `scipy` library. To see what would happen after .5 time units:

Python input

```

76 import scipy.linalg
77
78 print(scipy.linalg.expm(Q * 0.5).round(5))

```

which gives:

Python output

```

79 [[0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.0815 ]
80 [0.08501 0.18292 0.18666 0.1708  0.14377 0.1189  0.11194]
81 [0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346]
82 [0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142]
83 [0.02667 0.07361 0.10005 0.13422 0.17393 0.2189  0.27262]
84 [0.01567 0.0487  0.07552 0.11775 0.17512 0.24484 0.32239]
85 [0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914]]

```

To see what would happen after 500 time units:

Python input

```
86 print(scipy.linalg.expm(Q * 500).round(5))
```

which gives:

Python output

```
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]
91 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
92 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
93 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]]
```

No matter what state (row) the system is in, after 500 time units, the probability of ending up in each state (columns) is the same regardless of the state the system began in (row).

The analysis can in fact be stopped here however the choice of 500 time units was arbitrary and might not be the correct amount for all possible scenarios, as such the underlying equation 2.4 directly.

The underlying linear system will be solved using a numerically efficient algorithm called least squares optimisation (available from the **numpy** library):

Python input

```

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

```

The steady state vector for the default system is given by:

Python input

```

109 print(get_steady_state_vector(Q).round(5))

```

giving:

Python output

```

110 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]

```

This shows that the shop is expected to be empty approximately 3.4% of the time and full 26.2% of the time.

The final function written is one that uses all of the above to return the probability of the shop being full.

Python input

```

111 def get_probability_of_full_shop(
112     waiting_room=4, num_barbers=2
113 ):
114     """Return the probability of the barber shop being full.
115
116     Args:
117         waiting_room: an integer (default: 4)
118         num_barbers: an integer (default: 2)
119
120     Returns:
121         A real.
122     """
123     Q = get_transition_rate_matrix(
124         waiting_room=waiting_room,
125         num_barbers=num_barbers,
126     )
127     pi = get_steady_state_vector(Q)
128     return pi[-1]

```

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

Python input

```

129 print(round(get_probability_of_full_shop(), 6))

```

which gives:

Python output

```

130 0.261756

```

Now that the models have been defined, they will be used to compare the 2 possible scenarios.

Having 2 extra space in the waiting room corresponds to:

Python input

```
131 print(round(get_probability_of_full_shop(waiting_room=6), 6))
```

which gives:

Python output

```
132 0.23557
```

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

Python input

```
133 print(round(get_probability_of_full_shop(num_barbers=3), 6))
```

Python output

```
134 0.078636
```

Therefore, it would be better to increase the number of barbers by 1 than to increase the waiting room capacity by 2.

2.4 SOLVING WITH R

The first step taken is to write a function to obtain the transition rates between 2 given states:

R input

```

135 #' Return the transition rate for 2 given states.
136 #'
137 #' @param in_state an integer
138 #' @param out_state an integer
139 #' @param waiting_room an integer (default: 4)
140 #' @param num_barbers an integer (default: 2)
141 #'
142 #' @return A real
143 get_transition_rate <- function(in_state,
144                                out_state,
145                                waiting_room = 4,
146                                num_barbers = 2){
147
148   arrival_rate <- 10
149   service_rate <- 4
150
151   capacity <- waiting_room + num_barbers
152   delta <- out_state - in_state
153
154   if (delta == 1) {
155     if (in_state < capacity) {
156       return(arrival_rate)
157     }
158   }
159
160   if (delta == -1) {
161     return(min(in_state, num_barbers) * service_rate)
162   }
163   return(0)
164 }

```

This actual function will not be used but instead a vectorized version of this makes calculations more efficient:

R input

```

164 vectorized_get_transition_rate <- Vectorize(
165   get_transition_rate,
166   vectorize.args = c("in_state", "out_state")
167 )

```

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

```

168  #' Return the transition rate matrix Q
169  #'
170  #' @param waiting_room an integer (default: 4)
171  #' @param num_barbers an integer (default: 2)
172  #'
173  #' @return A matrix
174  get_transition_rate_matrix <- function(waiting_room = 4,
175                                       num_barbers = 2){
176    max_state <- waiting_room + num_barbers
177
178    Q <- outer(0:max_state,
179              0:max_state,
180              vectorized_get_transition_rate,
181              waiting_room = waiting_room,
182              num_barbers = num_barbers
183            )
184    row_sums <- rowSums(Q)
185
186    diag(Q) <- -row_sums
187    Q
188  }

```

Using this the matrix Q for the default system can be used:

R input

```

189  Q <- get_transition_rate_matrix()
190  print(Q)

```

which gives:

R output

```

191      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
192 [1,]  -10  10   0   0   0   0   0
193 [2,]   4 -14  10   0   0   0   0
194 [3,]   0   8 -18  10   0   0   0
195 [4,]   0   0   8 -18  10   0   0
196 [5,]   0   0   0   8 -18  10   0
197 [6,]   0   0   0   0   8 -18  10
198 [7,]   0   0   0   0   0   8 -8

```

One immediate thing that can be done with this matrix is to take the matrix exponential discussed above. To do this, an R library called `expm` will be used.

To be able to make use of the nice `%>%` “pipe” operator the `magrittr` library will be loaded. Now if to see what would happen after .5 time units:

R input

```

199 library(expm, warn.conflicts = FALSE, quietly = TRUE)
200 library(magrittr, warn.conflicts = FALSE, quietly = TRUE)
201
202 print( (Q * .5) %>% expm %>% round(5))

```

which gives:

R output

```

203      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
204 [1,] 0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.08150
205 [2,] 0.08501 0.18292 0.18666 0.17080 0.14377 0.11890 0.11194
206 [3,] 0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346
207 [4,] 0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142
208 [5,] 0.02667 0.07361 0.10005 0.13422 0.17393 0.21890 0.27262
209 [6,] 0.01567 0.04870 0.07552 0.11775 0.17512 0.24484 0.32239
210 [7,] 0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914

```

After 500 time units:

R input

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

which gives:

R output

```
212      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
213 [1,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
214 [2,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
215 [3,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
216 [4,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
217 [5,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
218 [6,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
219 [7,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
```

No matter what state (row) the system is in, after 500 time units, the probability of ending up in each state (columns) is the same regardless of the state the system began in (row).

The analysis can in fact be stopped here however the choice of 500 time units was arbitrary and might not be the correct amount for all possible scenarios, as such the underlying equation 2.4 directly.

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

R input

```

220 library(pracma, warn.conflicts = FALSE, quietly = TRUE)
221
222 #' Return the steady state vector of any given continuous time
223 #' transition rate matrix
224 #'
225 #' @param Q a transition rate matrix
226 #'
227 #' @return A vector
228 get_steady_state_vector <- function(Q){
229   state_space_size <- dim(Q)[1]
230   A <- rbind(t(Q), 1)
231   b <- c(integer(state_space_size), 1)
232   mldivide(A, b)
233 }

```

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

The steady state vector for the default system is now given by:

R input

```

234 print(get_steady_state_vector(Q))

```

giving:

R output

```

235      [,1]
236 [1,] 0.03430888
237 [2,] 0.08577220
238 [3,] 0.10721525
239 [4,] 0.13401906
240 [5,] 0.16752383
241 [6,] 0.20940479
242 [7,] 0.26175598

```

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 return the probability of the shop being full.

R input

```

243 #' Return the probability of the barber shop being full
244 #'
245 #' @param waiting_room (default: 4)
246 #' @param num_barbers (default: 2)
247 #'
248 #' @return A real
249 get_probability_of_full_shop <- function(waiting_room = 4,
250                                         num_barbers = 2){
251   arrival_rate <- 10
252   service_rate <- 4
253   pi <- get_transition_rate_matrix(
254     waiting_room = waiting_room,
255     num_barbers = num_barbers
256   ) %>%
257     get_steady_state_vector()
258
259   capacity <- waiting_room + num_barbers
260   pi[capacity + 1]
261 }

```

This confirms the previous probability calculated probability of the shop being full:

R input

```

262 print(get_probability_of_full_shop())

```

which gives:

R output

```

263 [1] 0.261756

```

Now that the models have been defined, they will be used to compare the 2 possible scenarios.

Adding 2 extra spaces in the waiting rooms corresponds to:

R input

264

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

which decreases the probability of a full shop to:

R output

265

```
[1] 0.2355699
```

but adding another barber and chair:

R input

266

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

gives:

R output

267

```
[1] 0.0786359
```

Therefore, it would be better to increase the number of barbers by 1 than to increase the waiting room capacity by 2.

2.5 RESEARCH

TBA



Discrete Event Simulation

COMPLEX situations further compounded by randomness appear throughout daily lives. Examples include 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, a particular approach we are going to cover here is called Discrete Event Simulation.

3.1 TYPICAL PROBLEM

A bicycle repair shop would like reconfigure in order to guarantee that all bicycles processed 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, staffed by one member of staff who can inspect one bicycle at a time. On average an inspection takes around 3 minutes.
- Around 20% of bicycles do not need repair after inspection, and they are then ready for collection.
- Around 80% of bicycles go on to be repaired after inspection. These then wait in line outside the repair workshop, which is staffed 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.

An assumption of infinite capacity at the bicycle repair shop for waiting bicycles is made. The shop will hire an 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 aspects 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 linked together such as the bicycle shop a method to model this situation is *Discrete Event Simulation*.

Consider one probabilistic event, rolling a six sided die where 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 the die is rolled a large number of times, $\frac{1}{6}$ of those rolls would be expected to be a 1.

Consider a random process in which the actual values of the probability of events occurring are not known. 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 probability of obtaining a 1 on this die be estimated?

Rolling the weighted die once does not give much information. However due to a theorem called the law of large numbers, this die can be rolled a number of times and find the proportion of those rolls which gave a 1. The more times we roll the die, the closer this proportion approaches the actual value of the probability of obtaining a 1.

For a complex system such as the bicycle shop the goal is 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 obtain an exact value. So, like the weighted die, the system will be observed a number of times and the overall proportions of bicycles spending longer than 30 minutes in the shop will converge to the exact value. Unlike rolling a weighted die, it is 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 an additional member of staff, do not yet exist, so observing this would be costly in terms of money also. It is possible to build a virtual representation of this complex system on a computer, and observe a virtual day of work much more quickly and with much less cost, 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 with random numbers, these need to be generated.

Computers are deterministic, therefore true randomness is in itself a challenging mathematical problem. 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 the law of large numbers can be used. 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:

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

The bicycle repair shop is a system of interactions of 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 to the repair workshop,
- the time those bicycles spend being repaired.

As the simulation progresses these events will be generated, and will interact together as described in Section 9.1. 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 like the weighted die, running this simulation once does not give much information. The simulation can be run many times and to give an average proportion.

The process outlined above is a particular implementation of Monte Carlo simulation called *Discrete Event Simulation*, which is a generic term for generating pseudorandom numbers and observes the emergent interactions. In practice there are two main approaches to simulating complex probabilistic systems such as this one: *event scheduling* and *process based* simulation. It so happens that the main implementations in Python and R use each of these approaches respectively.

3.2.1 Event Scheduling Approach

When using the event scheduling approach, the ‘virtual representation’ of the system is the collection of facilities that the bicycles use, shown in Figure 3.1. Then the 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 when events occur this causes further events to occur in the future, either immediately or after a delay. 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, the ‘virtual representation’ of the system is 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 these 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 the Ciw library will be used in order to conduct Discrete Event Simulation in Python. Ciw uses the event scheduling approach, which means the system’s facilities are defined, and customers then interact with them.

In this case there are two facilities to define: the inspection desk and the repair workshop. For each of these the following need to be defined:

- 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 the time between consecutive arrivals will be assumed to follow an

exponential distribution, as will the service time. These are common assumptions for this sort of queueing system.

In Ciw, these are defined as part of a Network object, created using the `ciw.create_network` function. The function below creates a Network object that defines the system for a given set of parameters bicycle repair shop:

Python input

```

268 import ciw
269
270
271 def build_network_object(
272     num_inspectors=1,
273     num_repairers=2,
274 ):
275     """Returns a Network object that defines the repair shop.
276
277     Args:
278         num_inspectors: a positive integer (default: 1)
279         num_repairers: a positive integer (default: 2)
280
281     Returns:
282         a Ciw network object
283     """
284     arrival_rate = 15
285     inspection_rate = 20
286     repair_rate = 10
287     prob_need_repair = 0.8
288     N = ciw.create_network(
289         arrival_distributions=[
290             ciw.dists.Exponential(arrival_rate),
291             ciw.dists.NoArrivals(),
292         ],
293         service_distributions=[
294             ciw.dists.Exponential(inspection_rate),
295             ciw.dists.Exponential(repair_rate),
296         ],
297         number_of_servers=[num_inspectors, num_repairers],
298         routing=[[0.0, prob_need_repair], [0.0, 0.0]],
299     )
300     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

```

301 N = build_network_object()
302 print(N.number_of_nodes)

```

which gives:

Python output

```

303 2

```

Now that the system is defined a Simulation object can be created. Once this is built the simulation can be run, that is observe it for one virtual day. The following function does this:

Python input

```

304 def run_simulation(network, seed=0):
305     """Builds a simulation object and runs it for 8 time units.
306
307     Args:
308         network: a Ciw network object
309         seed: a float (default: 0)
310
311     Returns:
312         a Ciw simulation object after a run of the simulation
313     """
314     max_time = 8
315     ciw.seed(seed)
316     Q = ciw.Simulation(network)
317     Q.simulate_until_max_time(max_time)
318     return Q

```

Notice here a random seed is set. This is because there is randomness in running the simulation, setting a seed ensures reproducible results. 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.

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 the `pandas` library will be used:

Python input

```

319 import pandas as pd
320
321
322 def get_proportion(Q):
323     """Returns the proportion of bicycles spending over a given
324     limit at the repair shop.
325
326     Args:
327         Q: a Ciw simulation object after a run of the
328         simulation
329
330     Returns:
331         a real
332     """
333     limit = 0.5
334     inds = Q.nodes[-1].all_individuals
335     recs = pd.DataFrame(
336         dr for ind in inds for dr in ind.data_records
337     )
338     recs["total_time"] = (
339         recs["exit_date"] - recs["arrival_date"]
340     )
341     total_times = recs.groupby("id_number")["total_time"].sum()
342     return (total_times > limit).mean()

```

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

Python input

```
343 N = build_network_object()
344 Q = run_simulation(N)
345 p = get_proportion(Q)
346 print(round(p, 6))
```

This gives:

Python output

```
347 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 a number of times, and an average proportion taken. The following function returns an average proportion:

Python input

```

348 def get_average_proportion(num_inspectors=1, num_repairers=2):
349     """Returns the average proportion of bicycles spending over
350     a given limit at the repair shop.
351
352     Args:
353         num_inspectors: a positive integer (default: 1)
354         num_repairers: a positive integer (default: 2)
355
356     Returns:
357         a real
358     """
359     num_trials = 100
360     N = build_network_object(
361         num_inspectors=num_inspectors,
362         num_repairers=num_repairers,
363     )
364     proportions = []
365     for trial in range(num_trials):
366         Q = run_simulation(N, seed=trial)
367         proportion = get_proportion(Q=Q)
368         proportions.append(proportion)
369     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

```

370 p = get_average_proportion(num_inspectors=1, num_repairers=2)
371 print(round(p, 6))

```

which gives:

Python output

```

372 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: 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? First look the situation where the additional member of staff works at the inspection desk is considered:

Python input

```
373 p = get_average_proportion(num_inspectors=2, num_repairers=2)
374 print(round(p, 6))
```

which gives:

Python output

```
375 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

```
376 p = get_average_proportion(num_inspectors=1, num_repairers=3)
377 print(round(p, 6))
```

which gives:

Python output

```
378 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 that each bicycle's sequence of actions must be defined, 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

```

379 library(simmer)
380
381 #' Returns a simmer trajectory object outlining the bicycles
382 #' path through the repair shop
383 #'
384 #' @return A simmer trajectory object
385 define_bicycle_trajectories <- function() {
386   inspection_rate <- 20
387   repair_rate <- 10
388   prob_need_repair <- 0.8
389   bicycle <-
390     trajectory("Inspection") %>%
391     seize("Inspector") %>%
392     timeout(function() {
393       rexp(1, inspection_rate)
394     }) %>%
395     release("Inspector") %>%
396     branch(
397       function() (runif(1) < prob_need_repair),
398       continue = c(F),
399       trajectory("Repair") %>%
400         seize("Repairer") %>%
401         timeout(function() {
402           rexp(1, repair_rate)
403         }) %>%
404         release("Repairer"),
405       trajectory("Out")
406     )
407   return(bicycle)
408 }

```

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

```

409 #' Runs one trial of the simulation.
410 #'
411 #' @param bicycle a simmer trajectory object
412 #' @param num_inspectors positive integer (default: 1)
413 #' @param num_repairers positive integer (default: 2)
414 #' @param seed a float (default: 0)
415 #'
416 #' @return A simmer simulation object after one run of
417 #'         the simulation
418 run_simulation <- function(bicycle,
419                           num_inspectors = 1,
420                           num_repairers = 2,
421                           seed = 0) {
422   arrival_rate <- 15
423   max_time <- 8
424   repair_shop <-
425     simmer("Repair Shop") %>%
426     add_resource("Inspector", num_inspectors) %>%
427     add_resource("Repairer", num_repairers) %>%
428     add_generator("Bicycle", bicycle, function() {
429       rexp(1, arrival_rate)
430     })
431
432   set.seed(seed)
433   repair_shop %>% run(until = 8)
434   return(repair_shop)
435 }

```

Notice here a random seed is set. This is because there are elements of randomness when running the simulation, setting a seed ensures reproducible results. 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.

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, Simmer's `get_mon_arrivals()` function gives a data frame that can be manipulated:

R input

```

436 #' Returns the proportion of bicycles spending over 30
437 #' minutes in the repair shop
438 #'
439 #' @param repair_shop a simmer simulation object
440 #'
441 #' @return a float between 0 and 1
442 get_proportion <- function(repair_shop) {
443   limit <- 0.5
444   recs <- repair_shop %>% get_mon_arrivals()
445   total_times <- recs$end_time - recs$start_time
446   return(mean(total_times > 0.5))
447 }

```

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

R input

```

448 bicycle <- define_bicycle_trajectories()
449 repair_shop <- run_simulation(bicycle = bicycle)
450 print(get_proportion(repair_shop = repair_shop))

```

This piece of code gives

R output

```

451 [1] 0.1343284

```

meaning 13.43% 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 a number of times, and an average proportion taken. In order to do so, the following is a function that performs the above experiment over a number of trials, then finds an average proportion:

R input

```

452  #' Returns the average proportion of bicycles spending over
453  #' a given limit at the repair shop.
454  #'
455  #' @param num_inspectors positive integer (default: 1)
456  #' @param num_repairers positive integer (default: 2)
457
458  #' @return a float between 0 and 1
459 get_average_proportion <- function(num_inspectors = 1,
460                                   num_repairers = 2) {
461   num_trials <- 100
462   bicycle <- define_bicycle_trajectories()
463   proportions <- c()
464   for (trial in 1:num_trials) {
465     repair_shop <- run_simulation(
466       bicycle = bicycle,
467       num_inspectors = num_inspectors,
468       num_repairers = num_repairers,
469       seed = trial
470     )
471     proportion <- get_proportion(
472       repair_shop = repair_shop
473     )
474     proportions[trial] <- proportion
475   }
476   return(mean(proportions))
477 }

```

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

R input

```

478 print(
479   get_average_proportion(
480     num_inspectors = 1,
481     num_repairers = 2)
482 )

```

which gives:

R output

```
483 [1] 0.1635779
```

that is, on average 16.36% of bicycles will spend longer than 30 minutes at the repair shop.

Now consider the two possible future scenarios: 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? First consider the the situation where the additional member of staff works at the inspection desk:

R input

```
484 print(  
485   get_average_proportion(  
486     num_inspectors = 2,  
487     num_repairers = 2)  
488   )
```

which gives:

R output

```
489 [1] 0.04221602
```

that is 4.22% of bicycles.

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

R input

```
490 print(  
491   get_average_proportion(  
492     num_inspectors = 1,  
493     num_repairers = 3)  
494   )
```

which gives:

R output

```
495 [1] 0.1224761
```

that is 12.25% 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 HIGHLIGHTS



III

Dynamical Systems



Modelling with Differential Equations

SYSTEMS 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. This chapter 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 recovery rate. The cost 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 the objects of interest are the variable x over time t , and the rate at which x changes with t , its derivative $\frac{dx}{dt}$. The differential equation describing this will be of the form:

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

for some function f . In this case, the number of infected individuals will be denoted as I , which will implicitly mean that I is a function of time: $I = I(t)$, and the rate at which individuals recover will be denoted by α , then the differential equation that describes the above situation is:

$$\frac{dI}{dt} = -\alpha I \quad (4.2)$$

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

In this particular case, one such function is:

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

This is a solution because: $\frac{dI}{dt} = -\alpha e^{-\alpha t} = -\alpha I$.

However here $I(0) = 1$, whereas for this problem we know that at time $t = 0$ there are 100 infected individuals. In general there are many such functions that can satisfy a differential equation, known as a family of solutions. To know which particular solution is relevant to the situation, some sort of initial (also referred to as boundary) condition is required. Here this would be:

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

To evaluate the cost the sum of the values of that function over time is needed. Integration gives exactly this, so the cost would be:

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

where K is the cost per person per unit time.

In the upcoming sections code will be used to confirm to carry out the above efficiently so as to answer the original question.

4.3 SOLVING WITH PYTHON

The first step is to write a function to obtain the differential equation. The Python library SymPy is used which allows symbolic calculations.

Python input

```

496 import sympy as sym
497
498 t = sym.Symbol("t")
499 alpha = sym.Symbol("alpha")
500 I_0 = sym.Symbol("I_0")
501 I = sym.Function("I")
502
503
504 def get_equation(alpha=alpha):
505     """Return the differential equation.
506
507     Args:
508         alpha: a float (default: symbolic alpha)
509
510     Returns:
511         A symbolic equation
512     """
513     return sym.Eq(sym.Derivative(I(t), t), -alpha * I(t))

```

This gives an equation that defines the population change over time:

Python input

```

514 eq = get_equation()
515 print(eq)

```

which gives:

Python output

```

516 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:

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

A value of α can be passed if required:

Python input

```

517 eq = get_equation(alpha=1)
518 print(eq)

```

Python output

```

519 Eq(Derivative(I(t), t), -I(t))

```

Now a function will be written to obtain the solution to this differential with initial condition $I(0) = I_0$:

Python input

```

520 def get_solution(I_0=I_0, alpha=alpha):
521     """Return the solution to the differential equation.
522
523     Args:
524         I_0: a float (default: symbolic I_0)
525         alpha: a float (default: symbolic alpha)
526
527     Returns:
528         A symbolic equation
529     """
530     eq = get_equation(alpha=alpha)
531     return sym.dsolve(eq, I(t), ics={I(0): I_0})

```

This can verify the solution discussed previously:

Python input

```

532 sol = get_solution()
533 print(sol)

```

which gives:

Python output

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

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

SymPy itself can be used to verify the result, by taking the derivative of the right hand side of our solution.

Python input

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

which gives:

Python output

```
536 True
```

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

Python input

```
537 sol = get_solution(alpha=2, I_0=100)
538 print(sol)
```

which gives:

Python output

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

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

Python input

```

540 def get_cost(
541     I_0=I_0,
542     alpha=alpha,
543     cost_per_person=10,
544     cost_of_cure=0,
545 ):
546     """Return the cost.
547
548     Args:
549         I_0: a float (default: symbolic I_0)
550         alpha: a float (default: symbolic alpha)
551         cost_per_person: a float (default: 10)
552         cost_of_cure: a float (default: 0)
553
554     Returns:
555         A symbolic expression
556     """
557     I_sol = get_solution(I_0=I_0, alpha=alpha)
558     return (
559         sym.integrate(I_sol.rhs, (t, 0, sym.oo))
560         * cost_per_person
561         + cost_of_cure * I_0
562     )

```

The cost without purchasing the cure is:

Python input

```

563 I_0 = 100
564 alpha = 2
565 cost_without_cure = get_cost(I_0=I_0, alpha=alpha)
566 print(cost_without_cure)

```

which gives:

Python output

567 500

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

Python input

```

568 cost_of_cure = 5
569 cost_with_cure = get_cost(
570     I_0=I_0, alpha=2 * alpha, cost_of_cure=cost_of_cure
571 )
572 print(cost_with_cure)

```

which gives:

Python output

573 750

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

4.4 SOLVING WITH R

R has some capability for symbolic mathematics, however at the time of writing the options available are somewhat limited and/or not reliable. As such, in R the problem will be solved using a numerical integration approach. For an outline of the theory behind this approach see Chapter 5.

First write a function to give the derivative for a given value of I .

R input

```

574 #' Returns the numerical value of the derivative.
575 #'
576 #' @param t a set of time points
577 #' @param y a function
578 #' @param parameters the set of all parameters passed to y
579
580 #' @return a float
581 derivative <- function(t, y, parameters) {
582   with(as.list(c(y, parameters)), {
583     dIdt <- -alpha * I # nolint
584     list(dIdt) # nolint
585   })
586 }

```

For example, to see the value of the derivative when $I = 0$:

R input

```

587 derivative(t = 0, y = c(I = 100), parameters = c(alpha = 2))

```

This gives:

R output

```

588 [[1]]
589 [1] -200

```

Now the deSolve library will be used for solving differential equations numerically:

R input

```
590 library(deSolve) # nolint
591 #' Return the solution to the differential equation.
592 #'
593 #' @param times: a vector of time points
594 #' @param y_0: a float (default: 100)
595 #' @param alpha: a float (default: 2)
596
597 #' @return A vector of numerical values
598 get_solution <- function(times,
599                           y0 = c(I = 100),
600                           alpha = 2) {
601   params <- c(alpha = alpha)
602   ode(y = y0, times = times, func = derivative, parms = params)
603 }
```

This will return a sequence of time point and values of I at those time points. Using this we can compute the cost.

R input

```

604 #' Return the cost.
605 #'
606 #' @param I_0: a float (default: symbolic I_0)
607 #' @param alpha: a float (default: symbolic alpha)
608 #' @param cost_per_person: a float (default: 10)
609 #' @param cost_of_cure: a float (default: 0)
610 #' @param step_size: a float (default: 0.0001)
611 #' @param max_time: an integer (default: 10)
612
613 #' @return A numeric value
614 get_cost <- function(
615     I_0 = 100,
616     alpha = 2,
617     cost_per_person = 10,
618     cost_of_cure = 0,
619     step_size = 0.0001,
620     max_time = 10) {
621   times <- seq(0, max_time, by = step_size)
622   out <- get_solution(times,
623     y0 = c(I = I_0),
624     alpha = alpha
625   )
626   number_of_observations <- length(out[, "I"])
627
628   time_between_steps <- diff(out[, "time"])
629   area_under_curve <- sum(
630     time_between_steps *
631     out[-number_of_observations, "I"]
632   )
633   area_under_curve *
634     cost_per_person + cost_of_cure *
635     I_0
636 }

```

The cost without purchasing the cure is:

R input

```
637 alpha <- 2
638 cost_without_cure <- get_cost(alpha = alpha)
639 print(round(cost_without_cure))
```

which gives:

R output

```
640 [1] 500
```

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

R input

```
641 cost_of_cure <- 5
642 cost_with_cure <- get_cost(
643   alpha = 2 * alpha, cost_of_cure = cost_of_cure
644 )
645 print(round(cost_with_cure))
```

which gives:

R output

```
646 [1] 750
```

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

4.5 RESEARCH

TBA



Systems Dynamics

IN many situations systems are dynamical, in that the state or population of a number of entities or classes change according to the current state or population of the system. For example population dynamics, chemical reactions, and systems of macroeconomics. It is often useful to be able to predict how these systems will behave over time, though the rules that govern these changes may be complex, and are not necessarily solvable analytically. In these cases numerical methods and visualisation may be used, which is the focus of this chapter.

5.1 PROBLEM

Consider the following scenario, where a population of 3000 people are susceptible to infection by some disease. This population can be described by the following parameters:

- They have a birth rate b of 0.01 per day;
- They have a death rate d of 0.01 per day;
- For every infectious individual, the infection rate α is 0.3 per day;
- Infectious people recover naturally (and thus gain an immunity from the disease), at a recovery rate r of 0.02 per day;
- For each day an individual is infected, they must take medication which costs a public healthcare system £10 per day.

A vaccine is produced, that allows new born individuals to gain an immunity. This vaccine costs the public health care system a one-off cost of £220 per vaccine. The healthcare providers would like to know if achieving a vaccination rate v of 85% would be beneficial financially.

5.2 THEORY

The above scenario is called a compartmental model of disease, and can be shown in the stock and flow diagram in Figure 5.1.

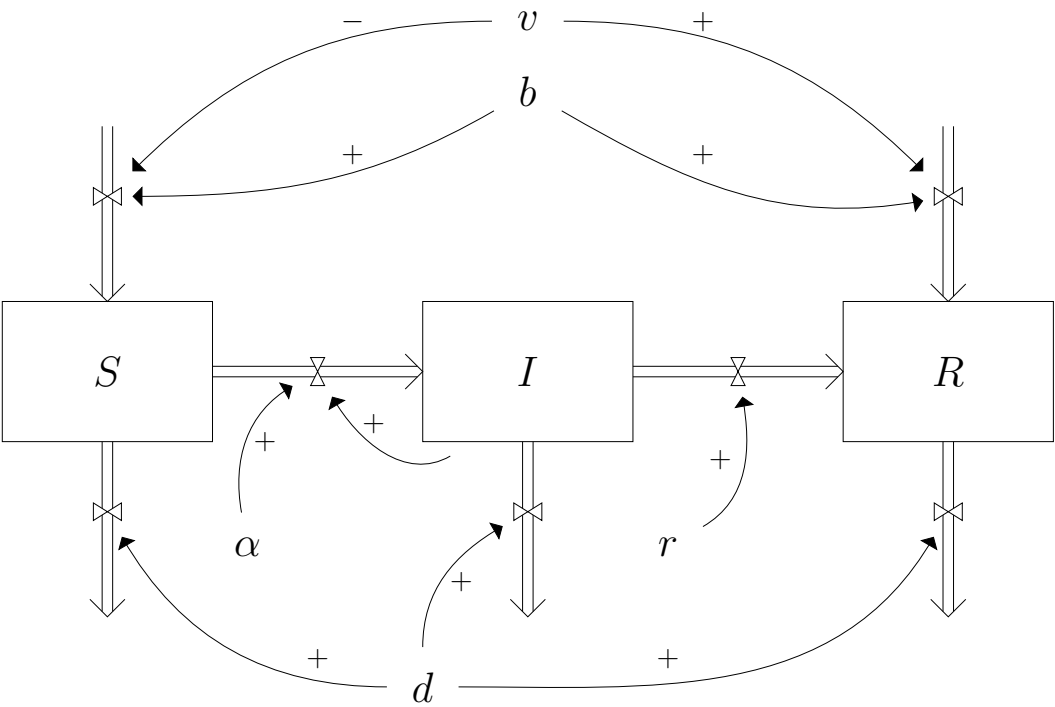


Figure 5.1 Diagrammatic representation of the epidemiology model

The system has three ‘stocks’ of different types of individuals, those susceptible to disease (S), those infected with the disease (I), and those who have recovered from the disease and so have gained immunity (R). The levels on these stocks change according to the flows in, out, and between them, controlled by ‘taps’. The amount of flow the taps let through are influenced in a multiplicative way (either negatively or positively), by other factors, such as external parameters (e.g. birth rate, infection rate) and the stock levels.

In this system the following taps exist, influenced by the following parameters:

- $external \rightarrow S$: Influenced positively by the birth rate, and negatively by the vaccine rate.
- $S \rightarrow I$: Influenced positively by the infection rate, and the number of infected individuals.
- $S \rightarrow external$: Influenced positively by the death rate.
- $I \rightarrow R$: Influenced positively by the recovery rate.
- $I \rightarrow external$: Influenced positively by the death rate.
- $R \rightarrow external$: Influenced positively by the birth rate and the vaccine rate.
- $external \rightarrow R$: Influenced positively by the death rate.

Mathematically the change in stock levels are written as the derivatives, for example the change in the number of susceptible individuals over time is denoted by $\frac{dS}{dt}$. This is equal to the sum of the taps in or out of that stock. Thus the system is described by the following system of differential equations:

$$\frac{dS}{dt} = -\frac{\alpha SI}{N} + (1-v)bN - dS \quad (5.1)$$

$$\frac{dI}{dt} = \frac{\alpha SI}{N} - (r+d)I \quad (5.2)$$

$$\frac{dR}{dt} = rI - dR + vbN \quad (5.3)$$

Where $N = S + I + R$ is the total number of individuals in the system.

We would like to understand the behaviour of the functions S , I and R under these rules, that is we would like to solve this system of differential equations. This system contains some non-linear terms, implying that this may be difficult to solve analytically, so we will use a numerical method instead.

There are a number of numerical methods, and the solvers we will use in Python and R cleverly choose the most appropriate for the problem at hand. In general methods for this kind of problems use the principle that the derivative denotes the rate of instantaneous change. Thus for a differential equation $\frac{dy}{dt} = f(t, y)$, consider the function y as a discrete sequence of points $\{y_0, y_1, y_2, y_3, \dots\}$ on $\{t_0, t_0 + h, t_0 + 2h, t_0 + 3h, \dots\}$ then

$$y_{n+1} = h \times f(t_0 + nh, y_n). \quad (5.4)$$

This sequence approaches the true solution y as $h \rightarrow 0$. Thus numerical methods, including the Runge-Kutta methods and the Euler method, step through this sequence $\{y_n\}$, choosing appropriate values of h and employing other methods of error reduction.

5.3 SOLVING WITH PYTHON

In this book we will use the `odeint` method of the SciPy library to numerically solve the above epidemiology models.

We first define the system of differential equations described in Equations 5.1, 5.2 and 5.3. This is a regular Python function, where the first two arguments are the system state and the current time respectively.

Python input

```

647 def derivatives(y, t, vaccine_rate, birth_rate=0.01):
648     """Defines the system of differential equations that
649 describe the epidemiology model.
650
651     Args:
652         y: a tuple of three integers
653         t: a positive float
654         vaccine_rate: a positive float <= 1
655         birth_rate: a positive float <= 1
656
657     Returns:
658         A tuple containing dS, dI, and dR
659     """
660     infection_rate = 0.3
661     recovery_rate = 0.02
662     death_rate = 0.01
663     S, I, R = y
664     N = S + I + R
665     dSdt = (
666         -((infection_rate * S * I) / N)
667         + ((1 - vaccine_rate) * birth_rate * N)
668         - (death_rate * S)
669     )
670     dIdt = (
671         ((infection_rate * S * I) / N)
672         - (recovery_rate * I)
673         - (death_rate * I)
674     )
675     dRdt = (
676         (recovery_rate * I)
677         - (death_rate * R)
678         + (vaccine_rate * birth_rate * N)
679     )
680     return dSdt, dIdt, dRdt

```

Using this function returns the instantaneous rate of change for each of the three stocks, S , I and R . If we begin at time 0.0, with 4 susceptible individuals, 1 infected individual, 0 recovered individuals, and a vaccine rate of 50%, then:

Python input

```
681 print(derivatives(y=(4, 1, 0), t=0.0, vaccine_rate=0.5))
```

Python output

```
682 (-0.255, 0.21, 0.045)
```

we would expect the number of susceptible individuals to reduce by around 0.255 per time unit, the number of infected individuals to increase by 0.21 per time unit, and the number of recovered individuals to increase by 0.045 per time unit. Now of course, after a tiny fraction of a time unit the stock levels will change, and thus the rates of change will change. So we will require something more sophisticated in order to determine the true behaviour of the system.

The following function observes the system's behaviour over some time period, using SciPy's `odeint` to numerically solve the system of differential equations:

Python input

```

683 from scipy.integrate import odeint
684
685
686 def integrate_ode(
687     derivative_function,
688     t,
689     y0=(2999, 1, 0),
690     vaccine_rate=0.85,
691     birth_rate=0.01,
692 ):
693     """Numerically solve the system of differential equations.
694
695     Args:
696         derivative_function: a function returning a tuple
697             of three floats
698         t: an array of increasing positive floats
699         y0: a tuple of three integers (default: (2999, 1, 0))
700         vaccine_rate: a positive float <= 1 (default: 0.85)
701         birth_rate: a positive float <= 1 (default: 0.01)
702
703     Returns:
704         A tuple of three arrays
705     """
706     results = odeint(
707         derivative_function,
708         y0,
709         t,
710         args=(vaccine_rate, birth_rate),
711     )
712     S, I, R = results.T
713     return S, I, R

```

Now we can use this function to investigate the difference in behaviour between a vaccination rate of 0% and a vaccination rate of 85%. Let's observe the system for two years, that is 730 days, in time steps of 0.01 days.

Begin with a vaccine rate of 0%:

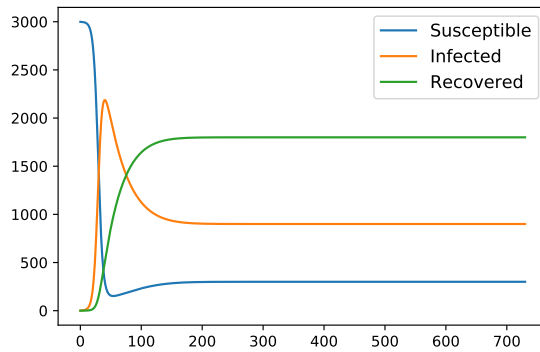


Figure 5.2 Output of code line 737-742

Python input

```

714 import numpy as np
715 from scipy.integrate import odeint
716
717 t = np.arange(0, 730.01, 0.01)
718 S, I, R = integrate_ode(derivatives, t, vaccine_rate=0.0)

```

Now S , I and R are arrays of values of the stock levels of S , I and R over the time steps t . Using `matplotlib` we can plot these to visualise their behaviour. The following code gives the plot shown in Figure 5.2.

Python input

```

719 import matplotlib.pyplot as plt
720
721 fig, ax = plt.subplots(1)
722 ax.plot(t, S, label='Susceptible')
723 ax.plot(t, I, label='Infected')
724 ax.plot(t, R, label='Recovered')
725 ax.legend(fontsize=12)
726 fig.savefig("plot_no_vaccine_python.pdf")

```

We observe that the number of infected individuals increases quickly, and in fact the rate of change increases as more individuals are infected. However this growth

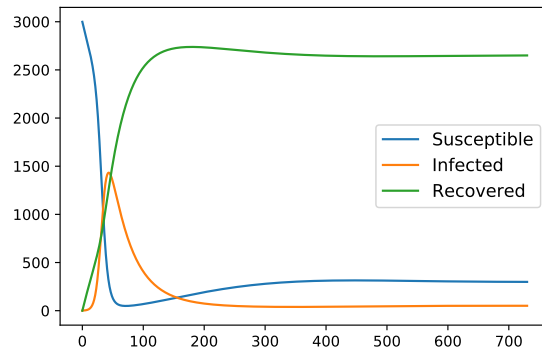


Figure 5.3 Output of code line 745-750

slows down as there are fewer susceptible individuals to infect. Due to the equal birth and death rates the overall population size remains constant; but we also see after some time period (around 300 time units) the levels of susceptible, infected, and recovered individuals becomes seemingly steady, and the disease becomes endemic. We can estimate once this steadiness occurs, around 10% of the population remain susceptible to the disease, 30% are infected, and 60% are recovered and immune.

Now with a vaccine rate of 85%:

Python input

```
727 t = np.arange(0, 730.01, 0.01)
728 S, I, R = integrate_ode(derivatives, t, vaccine_rate=0.85)
```

And again we can plot these to visualise their behaviour. The following code gives the plot shown in Figure 5.3.

Python input

```
729 fig, ax = plt.subplots(1)
730 ax.plot(t, S, label='Susceptible')
731 ax.plot(t, I, label='Infected')
732 ax.plot(t, R, label='Recovered')
733 ax.legend(fontsize=12)
734 fig.savefig("plot_with_vaccine_python.pdf")
```

With vaccination the disease remains endemic, however now we estimate that

once, steadiness occurs, around 10% of the population remain susceptible to the disease, 1.7% are infected, and 88.3% are immune or recovered and immune.

We've seen that vaccination lowers the percentage of the population living with the infection, which will lower the public healthcare system's medication costs. Let's now investigate if this saving is comparable to the cost of providing the vaccination to the newborns.

The following function calculates the total cost to the public healthcare system, that is the sum of the medication costs for those living with the infection and the vaccination costs:

Python input

```

735 def daily_cost(
736     derivative_function=derivatives, vaccine_rate=0.85
737 ):
738     """Calculates the daily cost to the public health system
739     after 2 years.
740
741     Args:
742         derivative_function: a function returning a tuple
743             of three floats
744         vaccine_rate: a positive float <= 1 (default: 0.85)
745
746     Returns:
747         the daily cost
748     """
749     max_time = 730
750     time_step = 0.01
751     birth_rate = 0.01
752     vaccine_cost = 220
753     medication_cost = 10
754     t = np.arange(0, max_time + time_step, time_step)
755     S, I, R = integrate_ode(
756         derivatives,
757         t,
758         vaccine_rate=vaccine_rate,
759         birth_rate=birth_rate,
760     )
761     N = S[-1] + I[-1] + R[-1]
762     daily_vaccine_cost = (
763         N * birth_rate * vaccine_rate * vaccine_cost
764     ) / time_step
765     daily_meds_cost = (I[-1] * medication_cost) / time_step
766     return daily_vaccine_cost + daily_meds_cost

```

Now let's compare the total daily cost with and without vaccination. Without vaccinations:

Python input

```

767 cost = daily_cost(vaccine_rate=0.0)
768 print(round(cost, 2))

```

which gives

Python output

```

769 900000.0

```

Therefore without vaccinations, once the infection is endemic, the public health care system would expect to spend £900,000 a day.

With a vaccine rate of 85%:

Python input

```

770 cost = daily_cost(vaccine_rate=0.85)
771 print(round(cost, 2))

```

which gives

Python output

```

772 611903.36

```

So vaccinating 85% of newborns would cost the public health care system, once the infection is endemic £611,903.36 a day. That is a saving of around 32%.

5.4 SOLVING WITH R

In this book we will use the `deSolve` library to numerically solve the above epidemiology models.

We first define the system of differential equations described in Equations 5.1, 5.2 and 5.3. This is an R function where the arguments are the current time, the system state, and a list of other parameters, respectively.

R input

```

773 #' Defines the system of differential equations that describe
774 #' the epidemiology model.
775 #'
776 #' @param t a positive float
777 #' @param y a tuple of three integers
778 #' @param vaccine_rate a positive float <= 1
779 #' @param birth_rate a positive float <= 1
780 #'
781 #' @return a list containing dS, dI, and dR
782 derivatives <- function(t, y, parameters){
783   infection_rate <- 0.3
784   recovery_rate <- 0.02
785   death_rate <- 0.01
786   with(as.list(c(y, parameters)), {
787     N <- S + I + R
788     dSdt <- ( - ( (infection_rate * S * I) / N) # nolint
789               + ( (1 - vaccine_rate) * birth_rate * N)
790               - (death_rate * S))
791     dIdt <- ( ( (infection_rate * S * I) / N) # nolint
792               - (recovery_rate * I)
793               - (death_rate * I))
794     dRdt <- ( (recovery_rate * I) # nolint
795               - (death_rate * R)
796               + (vaccine_rate * birth_rate * N))
797     list(c(dSdt, dIdt, dRdt)) # nolint
798   })
799 }

```

Using this function returns the instantaneous rate of change for each of the three stocks, S , I and R . If we begin at time 0.0, with 4 susceptible individuals, 1 infected individual, 0 recovered individuals, a vaccine rate of 50% and a birth rate of 0.01, then:

R input

```
800 derivatives(t = 0,  
801           y = c(S = 4, I = 1, R = 0),  
802           parameters = c(vaccine_rate = 0.5,  
803                         birth_rate = 0.01)  
804 )
```

R output

```
805 [[1]]  
806 [1] -0.255  0.210  0.045
```

we would expect the number of susceptible individuals to reduce by around 0.255 per time unit, the number of infected individuals to increase by 0.21 per time unit, and the number of recovered individuals to increase by 0.045 per time unit. Now of course, after a tiny fraction of a time unit the stock levels will change, and thus the rates of change will change. So we will require something more sophisticated in order to determine the true behaviour of the system.

The following function observes the system's behaviour over some time period, using the `deSolve` library to numerically solve the system of differential equations:

R input

```

807 library(deSolve) # nolint
808
809 #' Numerically solve the system of differential equations
810 #'
811 #' @param t an array of increasing positive floats
812 #' @param y0 list of integers (default: c(S=2999, I=1, R=0))
813 #' @param birth_rate a positive float <= 1 (default: 0.01)
814 #' @param vaccine_rate a positive float <= 1 (default: 0.85)
815 #'
816 #' @return a matrix of times, S, I and R values
817 integrate_ode <- function(times,
818                             y0 = c(S = 2999, I = 1, R = 0),
819                             birth_rate = 0.01,
820                             vaccine_rate = 0.84){
821   params <- c(birth_rate = birth_rate,
822               vaccine_rate = vaccine_rate)
823   ode(y = y0,
824       times = times,
825       func = derivatives,
826       parms = params)
827 }

```

Now we can use this function to investigate the difference in behaviour between a vaccination rate of 0% and a vaccination rate of 85%. Let's observe the system for two years, that is 730 days, in time steps of 0.01 days.

Begin with a vaccine rate of 0%:

R input

```

828 times <- seq(0, 730, by = 0.01)
829 out <- integrate_ode(times, vaccine_rate = 0.0)

```

Now `out`, is a matrix with four columns, `time`, `S`, `I` and `R`, which are arrays of values of the time points, and the stock levels of `S`, `I` and `R` over the time respectively. We can plot these to visualise their behaviour. The following code gives the plot shown in Figure 5.4.

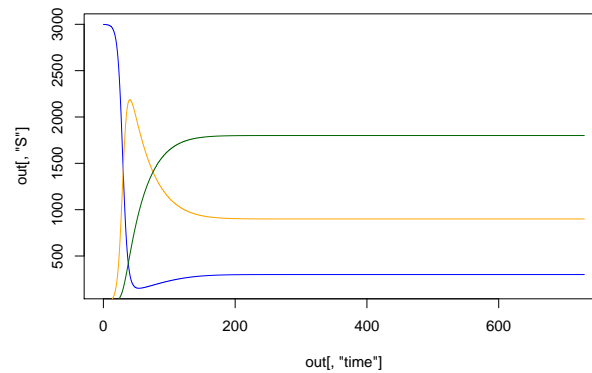


Figure 5.4 Output of code line 846-850

R input

```

830 pdf("plot_no_vaccine_R.pdf", width = 7, height = 5)
831 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
832 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
833 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
834 dev.off()

```

We observe that the number of infected individuals increases quickly, and in fact the rate of change increases as more individuals are infected. However this growth slows down as there are fewer susceptible individuals to infect. Due to the equal birth and death rates the overall population size remains constant; but we also see after some time period (around 300 time units) the levels of susceptible, infected, and recovered individuals becomes seemingly steady, and the disease becomes endemic. We can estimate once this steadiness occurs, around 10% of the population remain susceptible to the disease, 30% are infected, and 60% are recovered and immune.

Now with a vaccine rate of 85%:

R input

```

835 times <- seq(0, 730, by = 0.01)
836 out <- integrate_ode(times, vaccine_rate = 0.85)

```

And again we can plot these to visualise their behaviour. The following code gives the plot shown in Figure 5.5.

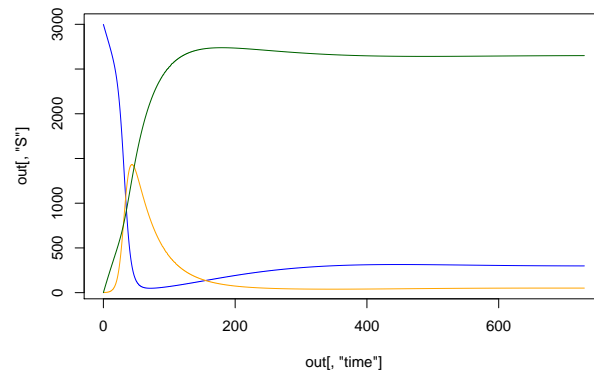


Figure 5.5 Output of code line 853-857

R input

```

837 pdf("plot_with_vaccine_R.pdf", width = 7, height = 5)
838 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
839 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
840 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
841 dev.off()

```

With vaccination the disease remains endemic, however now we estimate that once, steadiness occurs, around 10% of the population remain susceptible to the disease, 1.7% are infected, and 88.3% are immune or recovered and immune.

We've seen that vaccination lowers the percentage of the population living with the infection, which will lower the public healthcare system's medication costs. Let's now investigate if this saving is comparable to the cost of providing the vaccination to the newborns.

The following function calculates the total cost to the public healthcare system, that is the sum of the medication costs for those living with the infection and the vaccination costs:

R input

```

842 #' Calculates the daily cost to the public health
843 #' system after 2 years
844 #'
845 #' @param derivative_function: a function returning a
846 #'                               list of three floats
847 #' @param vaccine_rate: a positive float <= 1 (default: 0.85)
848 #'
849 #' @return the daily cost
850 daily_cost <- function(derivative_function = derivatives,
851                        vaccine_rate = 0.85){
852   max_time <- 730
853   time_step <- 0.01
854   birth_rate <- 0.01
855   vaccine_cost <- 220
856   medication_cost <- 10
857   times <- seq(0, max_time, by = time_step)
858   out <- integrate_ode(times, vaccine_rate = vaccine_rate)
859   N <- sum(tail(out[, c("S", "I", "R")], n = 1))
860   daily_vaccine_cost <- (N
861                         * birth_rate
862                         * vaccine_rate
863                         * vaccine_cost) / time_step
864   daily_medication_cost <- ( (tail(out[, "I"], n = 1)
865                             * medication_cost)) / time_step
866   daily_vaccine_cost + daily_medication_cost
867 }

```

Now let's compare the total daily cost with and without vaccination. Without vaccinations:

R input

```

868 cost <- daily_cost(vaccine_rate = 0.0)
869 print(cost)

```

which gives

R output

```
870 [1] 9e+05
```

Therefore without vaccinations, once the infection is endemic, the public health care system would expect to spend £900,000 a day.

With a vaccine rate of 85%:

R input

```
871 cost <- daily_cost(vaccine_rate = 0.85)
872 print(cost)
```

which gives

R output

```
873 [1] 611903.4
```

So vaccinating 85% of newborns would cost the public health care system, once the infection is endemic £611,903.40 a day. That is a saving of around 32%.

5.5 RESEARCH



IV

Emergent Behaviour



Game Theory

MOST when modelling certain situations two approaches are valid: to make assumptions about the overall behaviour or to make assumptions about the detailed behaviour. The latter falls is akin to measuring emergent behaviour. One tool used to do this is the study of interactive decision making: Game Theory.

6.1 PROBLEM

Consider a city council. Two electric taxi companies are going to move in to the city and the city wants to ensure that the customers are best served by this new duopoly. The two taxi firms will be deciding how many vehicles to deploy: one, two or three. The city wants to encourage them to both use three as this ensures the highest level of availability to the population.

Some exploratory data analysis gives the following insights:

- If both companies use the same number of taxis then they make the same profit which will go down slightly as the number of taxis goes up.
- If one company uses more taxis than the other then they make more profit.

The expected profits are given in Table 6.1.

Taxi numbers	Other company taxi numbers		
	1	2	3
1	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{1}{3}$
2	$\frac{3}{2}$	$\frac{19}{20}$	$\frac{1}{2}$
3	$\frac{5}{3}$	$\frac{4}{5}$	$\frac{17}{20}$

Table 6.1 Profits (in GBP per hour) of a given company based on their vehicle numbers and the other companies vehicle numbers.

Given these expected profits, the council wants to understand what is likely to happen and potentially give a financial incentive to each company to ensure their behaviour is in the population's interest.

The mathematical tool used to find the expected behaviour is Game Theory.

6.2 THEORY

In the case of this City, the interaction can be modelled using a mathematical object called a game which in the field of game theory is defined as follows. There are a number of games, the ones we will consider here require:

1. A given collection of actors that make decisions (players).
2. Options available to each player (actions).
3. A numerical value associated to each player for every possible choice of action made by all the players. This is the utility or reward.

There are called normal form games and are formally defined by:

1. A finite set of N players;
2. Action spaces for each player: $\{A_1, A_2, A_3, \dots, A_N\}$;
3. Utility functions that for each player $u_1, u_2, u_3, \dots, u_N$ where $u_i : A_1 \times A_2 \times A_3 \dots A_N \rightarrow \mathbb{R}$.

When $N = 2$ the utility function is often represented by a pair of matrices (1 for each player) of with the same number of rows and columns. The rows correspond to the actions available to the first player and the columns to the actions available to the second player.

Given a pair of actions (a row and column) we can read the utilities to both player by looking at the corresponding entry of the corresponding matrix.

A strategy corresponds to an way of choosing actions, this is represented by a probability vector. For the i th player, this vector v would be of size $|A_i|$ (the size of the action space) and v_i corresponds to the probability of choosing the i th action.

For the example of our City, the two matrices would be:

$$M = \begin{pmatrix} 1 & 1/2 & 1/3 \\ 3/2 & 19/20 & 1/2 \\ 5/3 & 4/5 & 17/20 \end{pmatrix} \quad N = M^T = \begin{pmatrix} 1 & 3/2 & 5/3 \\ 1/2 & 19/20 & 1/2 \\ 1/3 & 4/5 & 17/20 \end{pmatrix}$$

A diagram of the system is shown in Figure 6.1

Both taxis always choosing to use 2 taxis (the second row/column) would correspond to the strategy: $(0, 1, 0)$. If the both companies use this strategy and the row player (who controls the rows) wants to improve their outcome it's evident by inspecting the second column that the highest number is $19/20$: thus the row player has no reason to change what they are doing.

This is in fact called a Nash equilibrium: when both players are playing a strategy that is the best response against the other.

Whilst a Nash equilibria is not necessarily a set of strategies that players will converge towards, once they are there they have no reason to move away from it. It is the particular concept we will use to understand the emergent behaviour in our city.

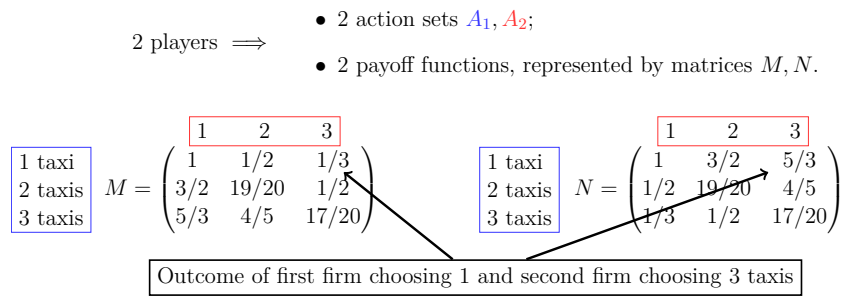


Figure 6.1 Diagrammatic representation of the action sets and payoff matrices for the game.

6.3 SOLVING WITH PYTHON

The first step we will take is to write a function to create a game using the matrix expected profits. We will use the `nashpy` library for this.

Python input

```

874 import nashpy as nash
875
876
877 def get_game(profits):
878     """Return the game object.
879
880     Args:
881         profits: a matrix with expected profits
882
883     Returns:
884         A nashpy game object
885     """
886     return nash.Game(profits, profits.T)

```

Using this we can obtain the game for the our problem:

Python input

```

887 import numpy as np
888
889 profits = np.array(
890     (
891         (1, 1 / 2, 1 / 3),
892         (3 / 2, 19 / 20, 1 / 2),
893         (5 / 3, 4 / 5, 17 / 20),
894     )
895 )
896 game = get_game(profits=profits)
897 print(game)

```

which gives:

Python output

```

898 Bi matrix game with payoff matrices:
899
900 Row player:
901 [[1.          0.5          0.33333333]
902  [1.5         0.95         0.5        ]
903  [1.66666667 0.8          0.85        ]]
904
905 Column player:
906 [[1.          1.5          1.66666667]
907  [0.5         0.95         0.8        ]
908  [0.33333333 0.5          0.85        ]]

```

We can now use this to investigate what stable behaviours might emerge:

Python input

```

909 for eq in game.support_enumeration():
910     print(eq)

```

which gives:

Python output

```

911 (array([0., 1., 0.]), array([0., 1., 0.]))
912 (array([0., 0., 1.]), array([0., 0., 1.]))
913 (array([0. , 0.7, 0.3]), array([0. , 0.7, 0.3]))

```

We see that there are 3 Nash equilibria: 3 possible pairs of behaviour that the two companies might converge to.

- The first equilibria $((0, 1, 0), (0, 1, 0))$ corresponds to both firms always using 2 taxis.
- The second equilibria $((0, 0, 1), (0, 0, 1))$ corresponds to both firms always using 3 taxis.
- The third equilibria $((0, 0.7, 0.3), (0, 0.7, 0.3))$ corresponds to both firms using 2 taxis 70% of the time and 3 taxis otherwise.

A good thing to note is that the two taxi companies will never only provide a single taxi (which would be harmful to the customers).

However, the Council would like to offset the cost of 3 taxis so as to encourage the taxi company to provide a better service. This involves modifying the `get_game` function as follows:

Python input

```

914 def get_game(profits, offset):
915     """Return the game object with a given offset when 3 taxis
916     are provided.
917
918     Args:
919         profits: a matrix with expected profits
920         offset: a float
921
922     Returns:
923         A nashpy game object
924     """
925     new_profits = np.array(profits)
926     new_profits[2] += offset
927     return nash.Game(new_profits, new_profits.T)

```

we will write a function `get_equilibria` which will directly compute the equilibria:

Python input

```

928 def get_equilibria(profits, offset):
929     """Return the equilibria for a given offset when 3 taxis
930 are provided.
931
932     Args:
933         profits: a matrix with expected profits
934         offset: a float
935
936     Returns:
937         A nashpy game object
938     """
939     game = get_game(profits=profits, offset=offset)
940     return tuple(game.support_enumeration())

```

Using this we can obtain the number of equilibria for a given offset and stop when there is a single equilibria:

Python input

```

941 offset = 0
942 while len(get_equilibria(profits=profits, offset=offset)) > 1:
943     offset += 0.01

```

This gives a final offset value of:

Python input

```

944 print(round(offset, 2))

```

Python output

```

945 0.15

```

and we can confirm that the Nash equilibria is where both taxi firms provide three vehicles:

Python input

```
946 print(tuple(get_equilibria(profits=profits, offset=offset)))
```

giving:

Python output

```
947 ((array([0., 0., 1.]), array([0., 0., 1.])),)
```

6.4 SOLVING WITH R

R does not have a single appropriate library for the game considered here, we will choose to use **Recon** which has functionality for finding the Nash equilibria for two player games when only considering pure strategies (where the players only choose to use a single action at a time).

R input

```
948 library(Recon)
949
950 #' Returns the equilibria in pure strategies
951 #'
952 #' @param profits: a matrix with expected profits
953 #'
954 #' @return a list of equilibria
955 get_equilibria <- function(profits){
956     sim_nasheq(profits, t(profits))
957 }
```

Using this we can obtain the pure Nash equilibria:

R input

```

958 profits <- rbind(
959     c(1, 1 / 2, 1 / 3),
960     c(3 / 2, 19 / 20, 1 / 2),
961     c(5 / 3, 4 / 5, 17 / 20)
962 )
963 eqs <- get_equilibria(profits = profits)
964 print(eqs)

```

which gives:

R output

```

965 $`Equilibrium 1`
966 [1] "2" "2"
967
968 $`Equilibrium 2`
969 [1] "3" "3"

```

We see that there are 2 pure Nash equilibria: 2 possible pairs of behaviour that the two companies might converge to.

- The first equilibria $((0, 1, 0), (0, 1, 0))$ corresponds to both firms always using 2 taxis.
- The second equilibria $((0, 0, 1), (0, 0, 1))$ corresponds to both firms always using 3 taxis.

There is in fact a third Nash equilibria where both taxi firms use 2 taxis 70% of the time and 3 taxis the rest of the time but **Recon** is unable to find Nash equilibria with mixed behaviour for games with more than two strategies.

As an aside, if we remove the option of using a single taxi then **Recon** can give us all three equilibria by passing the `type = "mixed"` argument to `sim_nasheq`.

A good thing to note is that the two taxi companies will not only provide a single taxi (which would be harmful to the customers).

As discussed, the Council would like to offset the cost of 3 taxis so as to encourage the taxi company to provide a better service. This involves modifying the `get_equilibria` function as follows:

R input

```

970 #' Returns the equilibria in pure strategies
971 #' for a given offset
972 #'
973 #' @param profits: a matrix with expected profits
974 #' @param offset: a float
975 #'
976 #' @return a list of equilibria
977 get_equilibria <- function(profits, offset){
978   new_profits <- rbind(
979     profits[c(1, 2), ],
980     profits[3, ] + offset)
981   sim_nasheq(new_profits, t(new_profits))
982 }

```

Using this we can obtain the number of equilibria for a given offset and stop when there is a single equilibria:

R input

```

983 offset <- 0
984 while (length(
985   get_equilibria(profits = profits, offset = offset)
986 ) > 1){
987   offset <- offset + 0.01
988 }

```

This gives a final offset value of:

R input

```

989 print(round(offset, 2))

```

R output

```

990 [1] 0.15

```

and we can confirm that the Nash equilibria is where both taxi firms provide three vehicles:

R input

```
991 print(get_equilibria(profits = profits, offset = offset))
```

giving:

R output

```
992 $`Equilibrium 1`  
993 [1] "3" "3"
```

6.5 RESEARCH

TBA

Agent Based Simulation

SOMETIMES we can know a lot about individuals' behaviours and interactions, and would like to know about how a whole population of such individuals might behave. For example psychologists and economists may know a lot about how individual spenders and vendors behave in response to given stimuli, and we'd like to know how these stimuli might effect the macro-economy. Agent based simulation (or agent based modelling, or ABM) is a paradigm of thinking that allows such emergent population level behaviour to be investigated from individual rules and interactions.

7.1 PROBLEM

Consider a city populated by two kinds of household, for example a household might be fans of Cardiff City FC or Swansea City AFC. Each household has a preference for living close to households of the same kind, and will move houses around the city while their preferences are not satisfied. In this situation we are interested in how segregated does the city naturally get under these sorts of preferences.

7.2 THEORY

The model considered here is considered a 'classic' one for the paradigm of agent based simulation, and is usually called Schelling's segregation model. It features in Thomas Schelling's book 'Micromotives and Macrobehaviours', whose title neatly summarises the world view of agent based modelling: we know, understand, determine, or can control individual micromotives; and from this we'd like to observe and understand macrobehaviours.

In general an agent based model consists of two components, agents, and an environment:

- Agents are autonomous entities that will periodically choose to take one of a number of actions (including the option not to take an action). These are chosen in order to maximise that agent's own utility function.
- An environment contains a number of agents and defines how they are interconnected. The agents may be homogeneous or heterogeneous, and the inter-

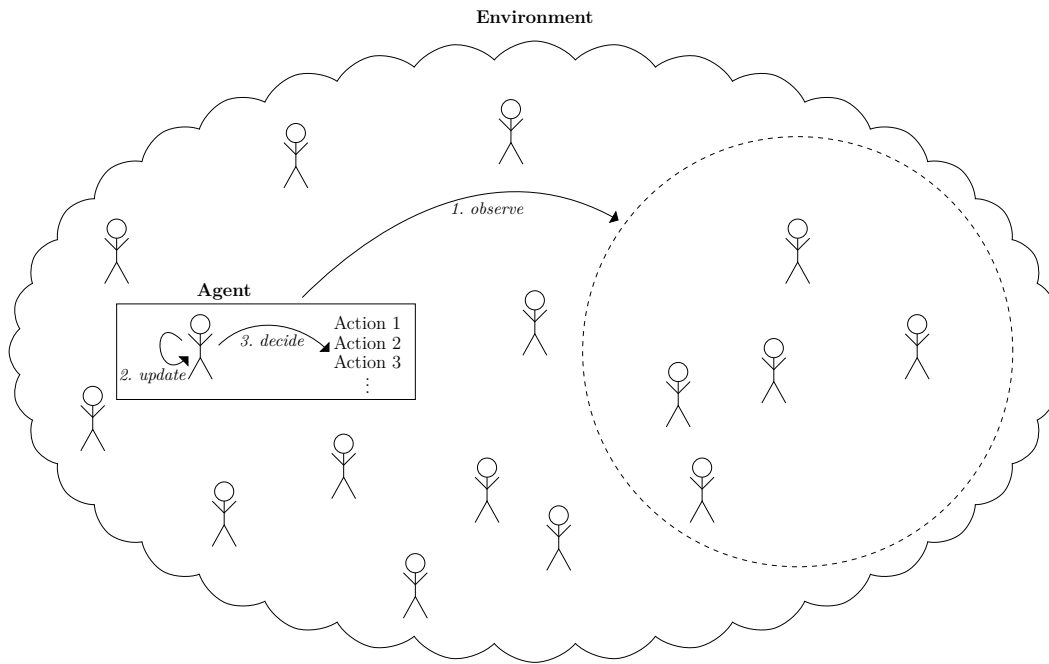


Figure 7.1 Representation of an agent interacting with its environment.

connections may change over time, possibly due to the actions taken by the agents.

In general, an agent will first observe a subset of its environment, for example it will consider some information about the agents it is currently interconnected with. Then it will update some information about itself based on these observations. This could be recording relevant information from the observations, but could also include some learning technique, maybe considering its own previous actions. It will then decide on an action to take, and carry out this action. This decision may be deterministic and rules-based, random, based on its own attributes from some learning process, or anything else; with the ultimate aim of maximising its own utility function. This process happens to all agents in the environment, possibly simultaneously. This is summarised in Figure 7.1

Notably, each agent is only behaving in a way that maximises its own utility function. Also, as each agent is part of every other agent's environment, then when the agents update themselves, and when the agents take actions, it can effect the behaviour of all other agents.

Let's consider the football team supporters problem. Each household is an agent. The environment is the city. Each household's utility function is to satisfy their preference of living next to at least a given number of households supporting the same team as themselves. Their action choices are to move house or not to move house. The subset of the environment they observe is their own neighbours.

In order to investigate the system's behaviour, we will simulate the system. As

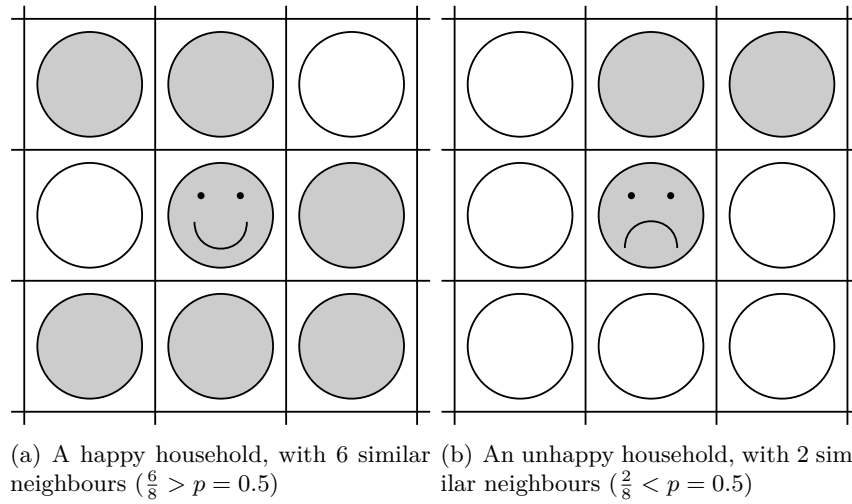


Figure 7.2 Example of a household happy and unhappy with its neighbours, when $p = 0.5$. Households supporting Cardiff City FC are shaded grey, households supporting Swansea City AFC are white.

a simplification we will model the city as a 50x50 grid. Each box is a house that can either contain a household of Cardiff City FC supporters, or contain a household of Swansea City AFC supporters. Define a house's neighbours by the grid locations adjacent to it, horizontally, vertically, and diagonally. For mathematical simplicity, also assume that the grid is a torus, where houses in the top row are vertically adjacent to the bottom row, and houses in the rightmost column are horizontally adjacent to the leftmost column.

Next let's consider each household's behaviour. Every household has a preference p . This corresponds to the minimum proportion of neighbours they are happy to live next to who support the same team as themselves. Figure 7.2 shows a household of Cardiff City FC supporters that are happy with their neighbours, and not happy with their neighbours, when $p = 0.5$. Households supporting Cardiff City FC are shaded grey, while households supporting Swansea City AFC are white.

The original problem stated that households randomly move around the city whenever they are unhappy with their neighbours. This long process of selling, searching for, and buying houses can be simplified to randomly pairing two unhappy households and swapping their houses. Let this happen to all unhappy households. In fact, we can simplify further and consider the houses themselves as agents, and who swap households with another house.

Therefore our model logic is:

1. Initialise the model: fill each house in the grid with either a household of Cardiff City FC or Swansea City AFC supporters with probability 0.5 each.
2. At each discrete time step, for every house:

- (a) Consider their household's neighbours (*observe*).
- (b) Determine if the household is happy (*update*).
- (c) If unhappy (*decide*), swap household with another randomly chosen house with an unhappy household (*action*).

After a number of time steps we can observe the overall structure of the city and any population level behaviour that may have emerged without explicit defining.

The above is an agent based model. It is a model as it is an abstraction of the real system. It is agent based as it only explicitly defines individual behaviours and interactions, but we wish to observe overall population level behaviours not explicitly defined. Note that this does not require code to analyse: in fact this model was originally run by placing and manually swapping silver and copper coins on a chessboard. A model isn't agent-based simply from the manner in which it is coded. Coding the model does however allow it to be run efficiently, scaled, and allows ease of analysis.

7.3 SOLVING WITH PYTHON

In agent based modelling we consider individual agents as their own entities, with their own rules and behaviours. This world view lends itself well to object-orientated programming. Here we build a number of *objects* from a set of instructions called a *class*. These objects can both store information (in Python we call these *attributes*), and do things (in Python we call these *methods*).

Python itself is written this way, and also allows users to define their own.

For this problem we will define two classes (types of object): a **House** and a **City** for them to live in.

First let's import some useful libraries:

```
Python input
994 import random
995 import itertools
996 import numpy as np
```

Now let's define the **City**:

Python input

```

997 class City:
998     def __init__(self, size, threshold):
999         """Initialises the City object.
1000
1001         Args:
1002             size: an integer number of rows and columns
1003             threshold: a number between 0 and 1 representing
1004                 the minimum acceptable proportion of similar
1005                 neighbours
1006         """
1007         self.size = size
1008         sides = range(size)
1009         self.coords = itertools.product(sides, sides)
1010         self.houses = {
1011             (x, y): House(x, y, threshold, self)
1012             for x, y in self.coords
1013         }
1014
1015     def run(self, n_steps):
1016         """Runs the simulation of a number of time steps.
1017
1018         Args:
1019             n_steps: an integer number of steps
1020         """
1021         for turn in range(n_steps):
1022             self.take_turn()
1023
1024     def take_turn(self):
1025         """Swaps all sad households."""
1026         sad = [h for h in self.houses.values() if h.sad()]
1027         random.shuffle(sad)
1028         i = 0
1029         while i <= len(sad) / 2:
1030             sad[i].swap(sad[-i])
1031             i += 1
1032
1033     def mean_satisfaction(self):
1034         """Finds the average household satisfaction.
1035
1036         Returns:
1037             The average city's household satisfaction
1038         """
1039         return np.mean(
1040             [h.satisfaction() for h in self.houses.values()]
1041         )

```

This defines a class, a template or a set of instructions that can be used to create instances of it, called objects. For our model we only need one instance of the `City` class, however it is useful to be able to produce more in order to run multiple trials with different random seeds. This class contains four methods: `__init__`, `run`, `take_turn` and `mean_satisfaction`.

The `__init__` method is run whenever the object is first created, and initialises the object. In this case it sets a number of attributes. First the square grid's `size` is defined, which is the number of rows and columns of houses it contains. Next we define `coords`, a list of tuples representing all the possible coordinates of the grid, this uses the `itertools` library for efficient looping. Finally `houses` is defined, a dictionary with grid coordinates as keys, and instances of the, yet to be defined, `House` class representing the houses themselves.

The `run` method runs the simulation. For each `n_steps` number of discrete time steps, the city runs the method `take_turn`. In this method, we first create a list of all the houses with households that are unhappy with their neighbours; these are put in a random order using the `random` library; and then working inwards from the ends, houses with sad households are paired up and swap households.

The last method defined here is the `mean_satisfaction` method, which is only used to observe any emergent behaviour. This calculates the average satisfaction of all the houses in the grid, using the `numpy` library for convenience.

In order to be able to create an instance of the above class, we need to define a `House` class:

Python input

```

1042 class House:
1043     def __init__(self, x, y, threshold, city):
1044         """Initialises the House object.
1045
1046         Args:
1047             x: the integer x-coordinate
1048             y: the integer y-coordinate
1049             threshold: a number between 0 and 1 representing
1050                 the minimum acceptable proportion of similar
1051                 neighbours
1052             city: an instance of the City class
1053         """
1054         self.x = x
1055         self.y = y
1056         self.threshold = threshold
1057         self.kind = random.choice(["Cardiff", "Swansea"])
1058         self.city = city
1059
1060     def satisfaction(self):
1061         """Determines the household's satisfaction level.
1062
1063         Returns:
1064             A proportion
1065         """
1066         same = 0
1067         for x, y in itertools.product([-1, 0, 1], [-1, 0, 1]):
1068             ax = (self.x + x) % self.city.size
1069             ay = (self.y + y) % self.city.size
1070             same += self.city.houses[ax, ay].kind == self.kind
1071         return (same - 1) / 8
1072
1073     def sad(self):
1074         """Determines if the household is sad.
1075
1076         Returns:
1077             a Boolean
1078         """
1079         return self.satisfaction() < self.threshold
1080
1081     def swap(self, house):
1082         """Swaps two households.
1083
1084         Args:
1085             house: the house object to swap household with
1086         """
1087         self.kind, house.kind = house.kind, self.kind

```

It contains four methods: `__init__`, `satisfaction`, `sad` and `swap`.

The `__init__` methods sets a number of attributes at the time the object is created: the house's `x` and `y` coordinates (its column and row numbers on the grid); its `threshold` which corresponds to p ; its `kind` which is randomly chosen between having a Cardiff City FC supporting household or a Swansea City AFC supporting household; and finally its `city`, an instance of the `City` class, shared by all the houses.

The `satisfaction` method loops through each of the house's neighbouring cells in the city grid, counts the number of neighbours that are of the same kind as itself, and returns this as a proportion. Then the `sad` method returns a boolean indicating of the household's satisfaction is below the minimum threshold.

Finally the `swap` method takes another house object, and swaps their household kinds.

Let's write a function that will let us create and run one of these simulations with a given random seed, threshold, and number of steps, and return the resulting mean happiness:

Python input

```

1088 def find_mean_happiness(seed, size, threshold, n_steps):
1089     """Create and run an instance of the simulation.
1090
1091     Args:
1092         seed: the random seed to use
1093         size: an integer number of rows and columns
1094         threshold: a number between 0 and 1 representing
1095             the minimum acceptable proportion of similar
1096             neighbours
1097         n_steps: an integer number of steps
1098
1099     Returns:
1100         The average city's household satisfaction after
1101         n_steps
1102     """
1103     random.seed(seed)
1104     C = City(size, threshold)
1105     C.run(n_steps)
1106     return C.mean_satisfaction()

```

Now let's run this for a city of size 50x50, with each household's threshold 0.65, and compare the mean happiness after 0 steps and 100 steps. First 0 steps:

Python input

```
1107 print(find_mean_happiness(0, 50, 0.65, 0))
```

Python output

```
1108 0.4998
```

This is well below the minimum threshold of 0.65, and so on average most households are unhappy here. Let's run it again for 100 generations and see how this changes:

Python input

```
1109 print(find_mean_happiness(0, 50, 0.65, 100))
```

Python output

```
1110 0.9078
```

After 100 time steps the average satisfaction level is much higher. In fact, is it much higher that each individual household's threshold. Now consider that this satisfaction level is really a level of how similar each households' neighbours are, it is actually a level of segregation. This was the central premise of Schelling's original model, that overall emergent segregation levels are much higher than any individuals' personal preference for segregation.

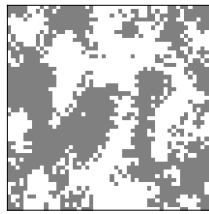
More analysis methods can be added, including plotting functions. Figure 7.3 shows the grid at the beginning, after 20 time steps, and after 100 time steps, with households supporting Cardiff City FC in grey, and those supporting Swansea City AFC in white. It visually shows the households naturally segregating over time.

7.4 SOLVING WITH R

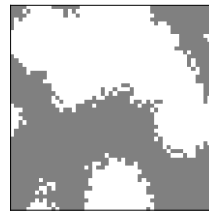
In agent based modelling we consider individual agents as their own entities, with their own rules and behaviours. This world view lends itself well to object-orientated programming. Here we build a number of *objects* from a set of instructions called a



(a) At the beginning.



(b) After 20 time steps.



(c) After 100 time steps.

Figure 7.3 Plotted results from the Python code.

class. These objects can both store information (in the R library we will use we call these *fields*), and do things (called *methods*).

There are a number of ways of doing object orientated programming in R. In this chapter, we will use a package called R6.

For this problem we will define two classes (types of object): a **House** and a **City** for them to live in.

Now let's define the **City**:

R input

```

1111 library(R6)
1112 city <- R6Class("City", list(
1113   size = NA,
1114   houses = NA,
1115   initialize = function(size, threshold) {
1116     self$size <- size
1117     self$houses <- c()
1118     for (x in 1:size) {
1119       row <- c()
1120       for (y in 1:size) {
1121         row <- c(row, house$new(x, y, threshold, self))
1122       }
1123       self$houses <- rbind(self$houses, row)
1124     } },
1125   run = function(n_steps) {
1126     if (n_steps > 0) {
1127       for (turn in 1:n_steps) {
1128         self$take_turn()
1129       } },
1130   take_turn = function() {
1131     sad <- c()
1132     for (house in self$houses) {
1133       if (house$sad()) {
1134         sad <- c(sad, house)
1135       } }
1136     sad <- sample(sad)
1137     num_sad <- length(sad)
1138     i <- 1
1139     while (i <= num_sad / 2) {
1140       sad[[i]]$swap(sad[[num_sad - i]])
1141       i <- i + 1
1142     } },
1143   mean_satisfaction = function() {
1144     mean(sapply(self$houses, function(x) x$satisfaction()))
1145   })
1146 )

```

This defines an R6 class, a template or a set of instructions that can be used to create instances of it, called objects. For our model we only need one instance of the `City` class, although it may be useful to be able to produce more in order to

run multiple trials with different random seeds. This class contains four methods: `initialize`, `run`, `take_turn` and `mean_satisfaction`.

The `initialize` method is run at the time the object is first created. It initialises the object by setting a number of its fields. First the square grid's `size` is defined, which is the number of rows and columns of houses it contains. Then it's `houses` is defined by iteratively repeating the `rbind` function to create a two-dimensional vector of instances of the, yet to be defined, `House` class, representing the houses themselves.

The `run` method runs the simulation. For each discrete time step from 1 to `n_steps`, the world runs the method `take_turn`. In this method, we first create a list of all the houses with households that are unhappy with their neighbours; these are put in a random order using the `sample` function; and then working inwards from the ends, houses with sad households are paired up and swap households.

The last method defined here is the `mean_satisfaction` method, which is used to observe any emergent behaviour. This calculates the average satisfaction of all the houses in the grid, using the `sapply` function to create a vector of all the houses' satisfaction levels.

In order to be able to create an instance of the above class, we need to define a `House` class:

R input

```

1147 house <- R6Class("House", list(
1148   x = NA,
1149   y = NA,
1150   threshold = NA,
1151   city = NA,
1152   kind = NA,
1153   initialize = function(x = NA,
1154                         y = NA,
1155                         threshold = NA,
1156                         city = NA) {
1157     self$x <- x
1158     self$y <- y
1159     self$threshold <- threshold
1160     self$city <- city
1161     self$kind <- sample(c("Cardiff", "Swansea"), 1)
1162   },
1163   satisfaction = function() {
1164     same <- 0
1165     for (x in -1:1) {
1166       for (y in -1:1) {
1167         ax <- ( (self$x + x - 1) %% self$city$size) + 1
1168         ay <- ( (self$y + y - 1) %% self$city$size) + 1
1169         if (self$city$houses[[ax, ay]]$kind == self$kind) {
1170           same <- same + 1
1171         } } }
1172     (same - 1) / 8
1173   },
1174   sad = function() {
1175     self$satisfaction() < self$threshold
1176   },
1177   swap = function(house) {
1178     old <- self$kind
1179     self$kind <- house$kind
1180     house$kind <- old
1181   })
1182 )

```

It contains four methods: `initialize`, `satisfaction`, `sad` and `swap`.

The `initialize` method sets a number of the class' fields when the object is created: the house's `x` and `y` coordinates (its column and row numbers on the grid); its `threshold` which corresponds to p ; its `kind` which is randomly chosen between

having a Cardiff City FC supporting household or a Swansea City AFC supporting household; and finally its `city`, an instance of the `City` class, shared by all the houses.

The `satisfaction` method loops through each of the house's neighbouring cells in the city grid, counts the number of neighbours that are of the same kind as itself, and returns this as a proportion. The `sad` method returns a boolean indicating of the household's satisfaction is below its minimum threshold.

Finally the `swap` method takes another house object, and swaps their household kinds.

Let's write a function that will let us create and run one of these simulations with a given random seed, threshold, and number of steps, and return the resulting mean happiness:

R input

```

1183  #' Create and run an instance of the simulation.
1184  #'
1185  #' @param seed: the random seed to use
1186  #' @param size: an integer number of rows and columns
1187  #' @param threshold: a number between 0 and 1 representing
1188  #'   the minimum acceptable proportion of similar neighbours
1189  #' @param n_steps: an integer number of steps
1190  #'
1191  #' @return The average city's household satisfaction
1192  #'   after n_steps
1193  find_mean_happiness <- function(seed, size,
1194                                threshold, n_steps){
1195    set.seed(seed)
1196    our_city <- city$new(size, threshold)
1197    our_city$run(n_steps)
1198    our_city$mean_satisfaction()
1199  }

```

Now let's run this for a city of size 50x50, with each household's threshold 0.65, and compare the mean happiness after 0 steps and 100 steps. First 0 steps:

R input

```

1200  print(find_mean_happiness(0, 50, 0.65, 0))

```


1201 **R output**

```
[1] 0.4956
```

This is well below the minimum threshold of 0.65, and so on average most households are unhappy here. Let's run the simulation for 100 generations and see how this changes:

1202 **R input**

```
print(find_mean_happiness(0, 50, 0.65, 100))
```

1203 **R output**

```
[1] 0.9338
```

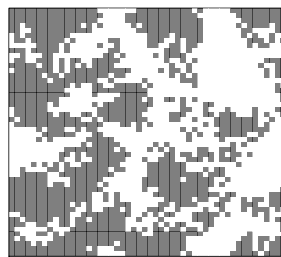
After 100 time steps the average satisfaction has increased. It is now actually much higher than each individual household's threshold. We can consider this satisfaction level as a level of how similar each household's neighbours are, and so it is actually a level of segregation. This was the central premise of Schelling's original model, that overall emergent segregation levels are much higher than any individuals' personal preference for segregation.

More analysis methods can be added, including plotting functions. Figure 7.4 shows the grid at the beginning, after 20 time steps, and after 100 time steps, with households supporting Cardiff City FC in grey, and those supporting Swansea City AFC in white. It visually shows the households naturally segregating over time.

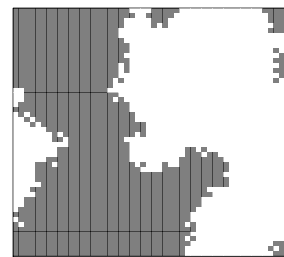
7.5 RESEARCH



(a) At the beginning.



(b) After 20 time steps.



(c) After 100 time steps.

Figure 7.4 Plotted results from the R code.

V

Optimisation



Linear Programming

FINDING the best configuration of some system can be challenging, especially when there is a seemingly endless amount of possible solutions. Optimisation techniques are a way to mathematically derive solutions that maximise or minimise some objective function, subject to a number of feasibility constraints. When all components of the problem can be written in a linear way, then linear programming is one technique that can be used to find the solution.

8.1 PROBLEM

A university runs 14 modules over three subjects: Art, Biology, and Chemistry. Each subject runs core modules and optional modules. Table 8.1 gives the module numbers for each of these.

The university is required to schedule examinations for each of these modules. The university would like the exams to be scheduled using the least amount of time slots possible. However not all modules can be scheduled at the same time as they share some students:

- All art modules share students,
- All biology modules share students,

Art Core	Biology Core	Chemistry Core
M00	M05	M09
M01	M06	M10
Art Optional	Biology Optional	Chemistry Optional
M02	M07	M11
M03	M08	M12
M04		M13

Table 8.1 List of modules on offer at the university.

- All chemistry modules share students,
- Biology students have the option of taking optional modules from chemistry, so all biology modules may share students with optional chemistry modules,
- Chemistry students have the option of taking optional modules from biology, so all chemistry modules may share students with optional biology modules,
- Biology students have the option of taking core art modules, and so all biology modules may share students with core art modules.

How can every exam be scheduled with no clashes, that using the least amount of time slots?

8.2 THEORY

Linear programming is a method that solves an optimisation problem of n variables by defining all constraints as planes in n -dimensional space. These planes combine to create a convex region where all feasible solutions (those that satisfy the constraints) lie within that region, and all infeasible solutions (those that break at least one constraint) lie outside that region.

We are interested in optimising, that is either minimising or maximising, some linear function, called the objective function. Therefore the solution must lie at the very edge of the feasible convex region, that is we have improved so much that if we were to improve any further we would lie outside the feasible region - hence the optimum lies on the edge.

Linear programming employs algorithms such as the Simplex method to mathematically traverse the edges of the feasible convex region, stopping at the optimum. Therefore to solve such a problem, we need to define out objective function and constraints in a linear fashion, and then apply appropriate algorithms.

Consider a 2-dimensional example: I am able to make £50 profit on each tonne of paint A I produce, and £60 profit on each tonne of paint B I produce. A tonne of paint A needs 4 tonnes of ingredient X and 5 tonnes of ingredient Y. A tonne of paint B needs 6 tonnes of ingredient X and 4 tonnes of ingredient Y. Only 24 tonnes of X and 20 tonnes of Y are available per day. How much of paint A and paint B should I produce daily to maximise profit?

This is formulated as a linear objective function, representing total profit, that is to be maximised; and two linear constraints, representing the availability of ingredients X and Y. They are written as:

$$\text{Maximise: } 50A + 60B \quad (8.1)$$

Subject to:

$$4A + 6B \leq 24 \quad (8.2)$$

$$5A + 4B \leq 20 \quad (8.3)$$

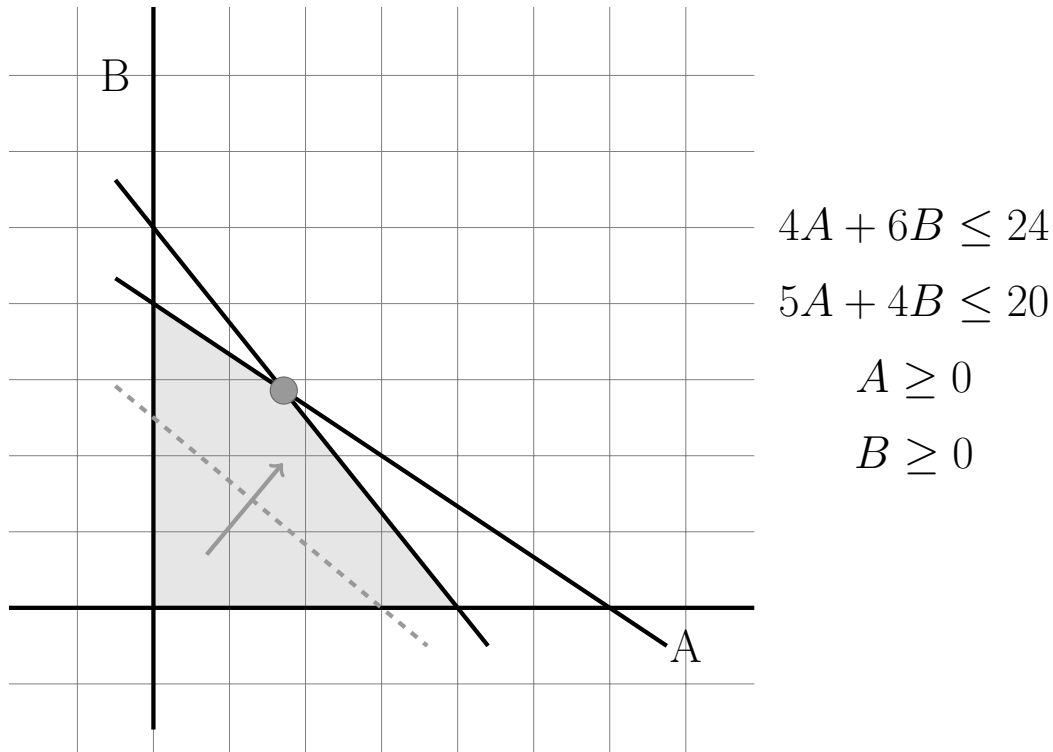


Figure 8.1 Visual representation of the paint linear program. The feasible convex region is shaded in grey; the objective function with arbitrary value is shown in a dashed line.

Now we have a linear system in 2-dimensional space with coordinates A and B . These are called the decision variables, whose values we wish to find that optimises the objective function given by expression 8.1. Inequalities 8.2 and 8.3 correspond to the amount of ingredient X and Y available per day. These, along with the additional constraints that we cannot produce a negative amount of paint ($A \geq 0$ and $B \geq 0$), form the convex feasible region shown in Figure 8.1.

Expression 8.1 corresponds to the total profit, which is the expression we are trying to maximise. As a line in the 2-dimensional space, this expression fixes its gradient, but its value determines the size of the y -intercept. Therefore optimising this function corresponds to pushing a line with that gradient to its furthest extreme within the feasible region, demonstrated in Figure 8.1. Therefore for this problem the optimum occurs in a particular vertex of the feasible region, at $A = \frac{12}{7}$ and $B = \frac{20}{7}$.

This works well as A and B can take any real value in the feasible region. It is common however to formulate Integer Linear Programmes where the decision variables are restricted to integers. There are a number of methods that can help us adapt a real solution to an integer solution. These include cutting planes, which introduce new constraints around the real solution to force an integer value; and branch and

bound methods, where we iteratively convert decision variables to their closest two integers and remove any infeasible solutions.

Both Python and R have libraries that carry out the linear and integer programming algorithms for us. When solving these kinds of problems, formulating them as linear systems is the most important challenge.

Consider again the exam scheduling problem from Section 9.1, and let's formulate this as a linear program. Define M as the set of all modules to be scheduled, and define T as the set of possible time slots. At worst each exam is scheduled for a different day, thus $|T| = |M| = 14$ in this case. Let $\{X_{mt} \text{ for } m \in M \text{ and } t \in T\}$ be a set of binary decision variables, that is $X_{mt} = 1$ if module m is scheduled for time t , and 0 otherwise.

There are six distinct sets of modules in which exams cannot be scheduled simultaneously: A_c, A_o representing core and optional art modules respectively; B_c, B_o representing core and optional biology modules respectively; and C_c, C_o representing core and optional chemistry modules respectively. Therefore $M = A_c \cup A_o \cup B_c \cup B_o \cup C_c \cup C_o$.

Additionally there are further clashes between these sets:

- No modules in $A_c \cup A_o$ can be scheduled together as they may share students, this is given by the constraint in inequality 8.7.
- No modules in $B_c \cup B_o \cup A_c$, can be scheduled together as they may share students, given by inequality 8.8.
- No modules in $B_c \cup B_o \cup C_o$, can be scheduled together as they may share students, given by inequality 8.9.
- No modules in $B_o \cup C_c \cup C_o$, can be scheduled together as they may share students, given by inequality 8.10.

Let's also define $\{Y_t \text{ for } t \in T\}$ as a set of auxiliary binary decision variables, where Y_t is 1 if time slot t is being used. This is enforced by Inequality 8.5.

Finally we have one final constraint, Equation 8.6, which ensures all modules are scheduled once and once only. Thus altogether our integer program becomes:

$$\text{Minimise: } \sum_{t \in T} Y_j \quad (8.4)$$

Subject to:

$$\frac{1}{|M|} \sum_{m \in M} X_{mt} \leq Y_j \text{ for all } j \in T \quad (8.5)$$

$$\sum_{t \in T} X_{mt} = 1 \text{ for all } m \in M \quad (8.6)$$

$$\sum_{m \in A_c \cup A_o} X_{mt} \leq 1 \text{ for all } t \in T \quad (8.7)$$

$$\sum_{m \in B_c \cup B_o \cup A_c} X_{mt} \leq 1 \text{ for all } t \in T \quad (8.8)$$

$$\sum_{m \in B_c \cup B_o \cup C_o} X_{mt} \leq 1 \text{ for all } t \in T \quad (8.9)$$

$$\sum_{m \in B_o \cup C_c \cup C_o} X_{mt} \leq 1 \text{ for all } t \in T \quad (8.10)$$

Another common way to define this linear program is by representing the coefficients of the constraints as a matrix. That is:

$$\text{Minimise: } c^T Z \quad (8.11)$$

Subject to:

$$AZ \star b \quad (8.12)$$

where Z is a vector representing the decision variables, c is the coefficients of the Z in the objective function, A is the matrix of the coefficients of Z in the constraints, b is the vector of the right hand side of the constraints, and \star represents either \leq , $=$ or \geq as required.

As Z is a one-dimensional vector of decisions variables, we ‘flatten’ the matrix X and the vector Y together to form this new variable. We can do this by first ordering by X then Y , within that ordering by time slot, then within that ordering by module number. Therefore:

$$Z_{|M|t+m} = X_{mt} \quad (8.13)$$

$$Z_{|M|^2+m} = Y_m \quad (8.14)$$

where t and m are indices starting at 0. For example Z_{17} would correspond to $X_{3,2}$, the decision variable representing whether module number 4 is scheduled on day 3; Z_{208} would correspond to Y_{12} , the decision variable representing whether there’s an exam scheduled for day 12.

Parameters c , A , and b can be determined by using this same conversion from the model in Equations 8.4 to 8.10. The vector c would be $|M|^2$ zeroes followed by $|M|$ ones. The vector b would be zeroes for all the rows representing Equation 8.5, and ones for all other constraints.

8.3 SOLVING WITH PYTHON

In this book we will use the Python library PuLP to formulate and solve the integer program. First let's define all the sets we will use to formulate the problem.

Python input

```

1204 Ac = [0, 1]
1205 Ao = [2, 3, 4]
1206 Bc = [5, 6]
1207 Bo = [7, 8]
1208 Cc = [9, 10]
1209 Co = [11, 12, 13]
1210 modules = Ac + Ao + Bc + Bo + Cc + Co
1211 times = range(14)

```

Now let's begin by defining an empty problem:

Python input

```

1212 import pulp
1213
1214 prob = pulp.LpProblem("ExamScheduling", pulp.LpMinimize)

```

We also need to define our sets of binary decision variables:

Python input

```

1215 xshape = (modules, times)
1216 x = pulp.LpVariable.dicts("X", xshape, cat=pulp.LpBinary)
1217 y = pulp.LpVariable.dicts("Y", times, cat=pulp.LpBinary)

```

Now y is a dictionary of binary decision variables, with keys as elements of the list `times`. Let's look at Y_3 corresponding to the third day:

Python input

```

1218 print(y[3])

```

Python output

1219 Y_3

While `x` is a dictionary of dictionaries of binary decision variables, with keys as elements of the lists `modules` and `times`. Let's look at $X_{2,5}$, the variable corresponding to module 2 being scheduled on day 5:

Python input

1220 `print(x[2][5])`

Python output

1221 X_2_5

Now we have an empty problem, all relevant sets, and all decision variables defined, we can go ahead and add the objective function and constraints to the problem.

For the objective function, Equation 8.4:

Python input

1222 `objective_function = sum([y[day] for day in times])`
 1223 `prob += objective_function`

Now the constraints, Inequalities 8.5-8.10:

Python input

```

1224 M = 1 / len(modules)
1225 for day in times:
1226     prob += M * sum(x[m][day] for m in modules) <= y[day]
1227     prob += sum([x[mod][day] for mod in Ac + Ao]) <= 1
1228     prob += sum([x[mod][day] for mod in Bc + Bo + Co]) <= 1
1229     prob += sum([x[mod][day] for mod in Bc + Bo + Ac]) <= 1
1230     prob += sum([x[mod][day] for mod in Cc + Co + Bo]) <= 1
1231
1232 for mod in modules:
1233     prob += sum(x[mod][day] for day in times) == 1

```

At this stage we could print the `prob` object, which would explicitly give all constraints written out fully. This can be used to error check if the need arises.

Now we can go ahead and solve the problem:

Python input

```

1234 prob.solve(pulp.apis.PULP_CBC_CMD(msg=False))

```

This method has also assigned values to our decision variables. These can be inspected, let's check if module 2 was scheduled for day 5:

Python input

```

1235 print(x[2][5].value())

```

Python output

```

1236 0.0

```

This was assigned the value 0, and so module 2 was not scheduled for that day. Let's check if module 2 was scheduled for day 9:

Python input

```
1237 print(x[2][9].value())
```

Python output

```
1238 1.0
```

This was assigned a value of 1, and so module 2 was scheduled for that day.

We can iterate through all decision variables and make a print solutions in order to read off the schedule easier:

Python input

```
1239 for day in times:
1240     if y[day].value() == 1:
1241         schedule = f"Day {day}: "
1242         for mod in modules:
1243             if x[mod][day].value() == 1:
1244                 schedule += f"{mod}, "
1245     print(schedule)
```

giving:

Python output

```
1246 Day 0: 1, 12,
1247 Day 5: 0, 13,
1248 Day 6: 11,
1249 Day 7: 4, 6, 10,
1250 Day 8: 3, 5, 9,
1251 Day 9: 2, 7,
1252 Day 13: 8,
```

Now the order of the days do not matter here, but we can see that 7 days are required in order to schedule all exams with no clashes, with two exams scheduled each day.

8.4 SOLVING WITH R

In R we will use the R package R0I, the R Optimization Infrastructure. This is a library of code that acts as a front end to a number of other solvers that need to be installed externally, allowing a range of optimisation problems to be solved with a number of different solvers, using similar problem structures and syntax. The solver that we will use here is called the CBC MILP Solver, which needs to be installed as well as the `rcbc` package.

The R0I package requires that the linear programme is represented in its matrix form, with a one-dimensional array of decision variables. Therefore we will use the form of the model described at the end of Section 9.2. We will write functions that define the objective function c , the coefficient matrix A , the vector of the right hand side of the constraints b , and the vector of equality or inequalities directions \star .

First we consider the objective function:

R input

```

1253 #' Writes the row of coefficients for the objective function
1254 #'
1255 #' @param n_modules: the number of modules to schedule
1256 #' @param n_days: the maximum number of days to schedule
1257 #'
1258 #' @return the objective function row to minimise
1259 write_objective <- function(n_modules, n_days){
1260   all_days <- rep(0, n_modules * n_days)
1261   Ys <- rep(1, n_days)
1262   append(all_days, Ys)
1263 }

```

For 3 modules and 3 days:

R input

```

1264 write_objective(3, 3)

```

Which gives the following array, corresponding the the coefficients of the array Z for Equation 8.4.

R output

```

1265 [1] 0 0 0 0 0 0 0 0 0 1 1 1

```

The following function is used to write one row of that coefficients matrix, for a given day, for a given set of clashes, corresponding to Inequalities 8.7 to 8.10:

R input

```

1266 #' Writes the constraint row dealing with clashes
1267 #'
1268 #' @param clashes: a vector of module indices that all cannot
1269 #'                  be scheduled at the same time
1270 #' @param day: an integer representing the day
1271 #'
1272 #' @return the constraint row corresponding to that set of
1273 #'         clashes on that day
1274 write_X_clashes <- function(clashes, day, n_days, n_modules){
1275   today <- rep(0, n_modules)
1276   today[clashes] = 1
1277   before_today <- rep(0, n_modules * (day - 1))
1278   after_today <- rep(0, n_modules * (n_days - day))
1279   all_days <- c(before_today, today, after_today)
1280   full_coeffs <- c(all_days, rep(0, n_days))
1281   full_coeffs
1282 }

```

where `clashes` is an array containing the module numbers of a set of modules that may all share students.

The following function is used to write one row of the coefficients matrix, for each module, ensuring that each module is scheduled on one day and one day only, corresponding to Equation 8.6:

R input

```

1283 #' Writes the constraint row to ensure that every module is
1284 #' scheduled once and only one
1285 #'
1286 #' @param module: an integer representing the module
1287 #'
1288 #' @return the constraint row corresponding to scheduling a
1289 #' module on only one day
1290 write_X_requirements <- function(module, n_days, n_modules){
1291   today <- rep(0, n_modules)
1292   today[module] = 1
1293   all_days <- rep(today, n_days)
1294   full_coeffs <- c(all_days, rep(0, n_days))
1295   full_coeffs
1296 }

```

The following function is used to write one row of the coefficients matrix corresponding to the auxiliary constraints of Inequality 8.5:

R input

```

1297 #' Writes the constraint row representing the Y variable,
1298 #' whether at least one exam is scheduled on that day
1299 #'
1300 #' @param day: an integer representing the day
1301 #'
1302 #' @return the constraint row corresponding to creating Y
1303 write_Y_constraints <- function(day, n_days, n_modules){
1304   today <- rep(1, n_modules)
1305   before_today <- rep(0, n_modules * (day - 1))
1306   after_today <- rep(0, n_modules * (n_days - day))
1307   all_days <- c(before_today, today, after_today)
1308   all_Ys <- rep(0, n_days)
1309   all_Ys[day] = -n_modules
1310   full_coeffs <- append(all_days, all_Ys)
1311   full_coeffs
1312 }

```

Finally the following function uses them all to assemble a coefficients matrix. It loops though the parameters for each constraint row required, uses the appropriate

function to create the row of the coefficients matrix, sets the appropriate inequality direction (\leq , $=$, \geq), and the value of the right hand side. It returns all three components:

R input

```

1313 #' Writes all the constraints as a matrix, column of
1314 #' inequalities, and right hand side column.
1315 #'
1316 #' @param list_clashes: a list of vectors with sets of modules
1317 #' that cannot be scheduled at the same time
1318 #'
1319 #' @return f.con the LHS of the constraints as a matrix
1320 #' @return f.dir the directions of the inequalities
1321 #' @return f.rhs the values of the RHS of the inequalities
1322 write_constraints <- function(list_clashes, n_days, n_modules){
1323   all_rows <- c()
1324   all_dirs <- c()
1325   all_rhss <- c()
1326   n_rows <- 0
1327
1328   for (clash in list_clashes){
1329     for (day in 1:n_days){
1330       clashes <- write_X_clashes(clash, day, n_days, n_modules)
1331       all_rows <- append(all_rows, clashes)
1332       all_dirs <- append(all_dirs, "<=")
1333       all_rhss <- append(all_rhss, 1)
1334       n_rows <- n_rows + 1
1335     }
1336   }
1337
1338   for (module in 1:n_modules){
1339     reqs <- write_X_requirements(module, n_days, n_modules)
1340     all_rows <- append(all_rows, reqs)
1341     all_dirs <- append(all_dirs, "==")
1342     all_rhss <- append(all_rhss, 1)
1343     n_rows <- n_rows + 1
1344   }
1345
1346   for (day in 1:n_days){
1347     Yconstraints <- write_Y_constraints(day, n_days, n_modules)
1348     all_rows <- append(all_rows, Yconstraints)
1349     all_dirs <- append(all_dirs, "<=")
1350     all_rhss <- append(all_rhss, 0)
1351     n_rows <- n_rows + 1
1352   }
1353
1354   f.con <- matrix(all_rows, nrow = n_rows, byrow = TRUE)
1355   f.dir <- all_dirs
1356   f.rhs <- all_rhss
1357   list(f.con, f.dir, f.rhs)
1358 }

```

For demonstration, if we had two modules and two possible days, with the single constraint that both modules cannot be scheduled at the same time, then:

R input

```
1359 write_constraints(list_clashes = list(c(1, 2)),
1360                  n_days = 2,
1361                  n_modules = 2)
```

This would give three components:

- a coefficient matrix of the left hand side of the constraints, A , (rows 1 and 2 corresponding to the clash on days 1 and 2, row 3 ensuring module 1 is scheduled on one day only, row 4 ensuring module 2 is scheduled on one day only, and rows 5 and 6 defining the decision variables Y),
- an array of direction of the constraint inequalities, \star ,
- and an array of the right hand side values of the constraints, b .

R output

```
1362 [[1]]
1363      [,1] [,2] [,3] [,4] [,5] [,6]
1364 [1,]    1    1    0    0    0    0
1365 [2,]    0    0    1    1    0    0
1366 [3,]    1    0    1    0    0    0
1367 [4,]    0    1    0    1    0    0
1368 [5,]    1    1    0    0   -2    0
1369 [6,]    0    0    1    1    0   -2
1370
1371 [[2]]
1372 [1] "<=" "<=" "==" "==" "<=" "<="
1373
1374 [[3]]
1375 [1] 1 1 1 1 0 0
```

Now we are ready to use these to solve the exam scheduling problem. First we define some parameters, including the sets of modules that all share students, that is the list of clashes:

R input

```

1376 n_modules = 14
1377 n_days = 14
1378
1379 Ac <- c(0, 1)
1380 Ao <- c(2, 3, 4)
1381 Bc <- c(5, 6)
1382 Bo <- c(7, 8)
1383 Cc <- c(9, 10)
1384 Co <- c(11, 12, 13)
1385
1386 list_clashes <- list(
1387   c(Ac, Ao),
1388   c(Bc, Bo, Co),
1389   c(Bc, Bo, Ac),
1390   c(Bo, Cc, Co)
1391 )

```

Then we can use the functions defined above to create the objective function and the three elements of the constraints:

R input

```

1392 constraints <- write_constraints(list_clashes = list_clashes,
1393                                n_days = n_days,
1394                                n_modules = n_modules)
1395 f.con <- constraints[[1]]
1396 f.dir <- constraints[[2]]
1397 f.rhs <- constraints[[3]]
1398 f.obj <- write_objective(n_modules = n_modules, n_days = n_days)

```

Finally, once these objects are in place, we can use the ROI library to construct an optimisation problem object:

R input

```

1399 library(ROI)
1400
1401 milp <- OP(objective = L_objective(f.obj),
1402            constraints = L_constraint(L = f.con,
1403                                     dir = f.dir,
1404                                     rhs = f.rhs),
1405            types = rep("B", length(f.obj)),
1406            maximum = FALSE)

```

This creates an `OP` object from our objective row `f.obj`, and our constraints which are made up from the three components `f.con`, `f.dir` and `f.rhs`. When creating this object we also denote the `types` as binary variables (an array of `"B"` for each decision variable), and we want to minimise the objective function so we set `maximum = FALSE`.

Now to solve:

R input

```
1407 | sol <- ROI_solve(milp)
```

The solver will output information about the solve process and runtime. We can now print the solution:

R input

```
1408 | print(sol$solution)
```

R output

[illegible]

This gives the values of each of the Z decision variables. We know the structure of this, that is the first 14 variables are the modules scheduled for day 1, and so on. The following code prints a readable schedule:

R input

```

1417 for (day in 1:n_days){
1418   if (sol$solution[(n_days * n_modules) + day] == 1){
1419     schedule <- paste("Day", day, ":")
1420     for (module in 1:n_modules){
1421       var <- ((day - 1) * n_modules) + module
1422       if (sol$solution[var] == 1){
1423         schedule <- paste(schedule, module)
1424       }
1425     }
1426     print(schedule)
1427   }
1428 }

```

R output

```

1429 [1] "Day 2 : 4 11"
1430 [1] "Day 6 : 1 12"
1431 [1] "Day 8 : 7"
1432 [1] "Day 10 : 8"
1433 [1] "Day 11 : 3 13"
1434 [1] "Day 12 : 2 6 9 14"
1435 [1] "Day 14 : 5 10"

```

This gives that 7 days are the minimum required to schedule the 14 exams without clashes, with either 1, 2 or 4 exams scheduled on each day.

8.5 RESEARCH

Heuristics

It is often necessary to find the most desirable choice from a large, or indeed, infinite set of options. Sometimes this can be done using exact techniques but often this is not possible and finding an almost perfect choice quickly is just as good. This is where the field of heuristics comes in to play.

9.1 PROBLEM

Consider a delivery company needs to deliver goods to 13 different stops. They need to find a route for a driver to traverse that stops at each of stops once only, then returns to the first stop, the depot.

The stops are represented spatially in Figure 9.2.

The relevant information is the pairwise distances between each of the stops, which is given by the distance matrix in equation (9.1).

$$d = \begin{bmatrix} 0 & 35 & 35 & 29 & 70 & 35 & 42 & 27 & 24 & 44 & 58 & 71 & 69 \\ 35 & 0 & 67 & 32 & 72 & 40 & 71 & 56 & 36 & 11 & 66 & 70 & 37 \\ 35 & 67 & 0 & 63 & 64 & 68 & 11 & 12 & 56 & 77 & 48 & 67 & 94 \\ 29 & 32 & 63 & 0 & 93 & 8 & 71 & 56 & 8 & 33 & 84 & 93 & 69 \\ 70 & 72 & 64 & 93 & 0 & 101 & 56 & 56 & 92 & 81 & 16 & 5 & 69 \\ 35 & 40 & 68 & 8 & 101 & 0 & 76 & 62 & 11 & 39 & 91 & 101 & 76 \\ 42 & 71 & 11 & 71 & 56 & 76 & 0 & 15 & 65 & 81 & 40 & 60 & 94 \\ 27 & 56 & 12 & 56 & 56 & 62 & 15 & 0 & 50 & 66 & 41 & 58 & 82 \\ 24 & 36 & 56 & 8 & 92 & 11 & 65 & 50 & 0 & 39 & 81 & 91 & 74 \\ 44 & 11 & 77 & 33 & 81 & 39 & 81 & 66 & 39 & 0 & 77 & 79 & 37 \\ 58 & 66 & 48 & 84 & 16 & 91 & 40 & 41 & 81 & 77 & 0 & 20 & 73 \\ 71 & 70 & 67 & 93 & 5 & 101 & 60 & 58 & 91 & 79 & 20 & 0 & 65 \\ 69 & 37 & 94 & 69 & 69 & 76 & 94 & 82 & 74 & 37 & 73 & 65 & 0 \end{bmatrix} \quad (9.1)$$

The value d gives the travel distance between stops i and j . For example, $d_{23} = 67$ indicates that the distance between the 2nd and 3rd stop in the route is 67.

The delivery company would like to find the route around the 13 stops that gives the smallest overall travel distance.

9.2 THEORY

This is an example of what is known as the Travelling Salesman Problem, and can be considered too complex to solve using exact methods. Heuristics are a family of methods that can be used to find a *sufficiently good* solution, though not necessarily the optimal solution, where the emphasis is on prioritising computational efficiency.

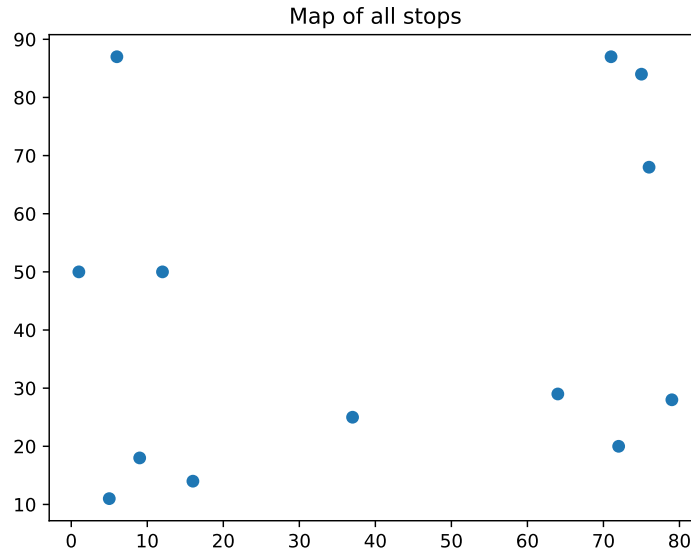


Figure 9.1 Spatial representation of the required stops.

The heuristic approach taken here will be to use a neighbourhood search algorithm. This algorithm works by considering a given potential solution, evaluating it and then trying another potential solution *close* to it. What *close* means depends on different approaches and problems: it is referred to as the neighbourhood. When new solution is considered *good* (this is again a term that depends on the approach and problem) then the search continues from the neighbourhood of this new solution.

For this problem, first represent a possible solution, that is a given route between all the potential stops as a *tour*. If there are 3 total stops and require that the tour starts and stops at the first one then there are two possible tours:

$$t \in \{(1, 2, 3, 1), (1, 3, 2, 1)\}$$

Given a distance matrix d such that d_{ij} is the distance between stop i and j the total cost of a tour is given by:

$$C(t) = \sum_{i=1}^n d_{t_i, t_{i+1}}$$

Thus, with:

$$d = \begin{pmatrix} 0 & 1 & 3 \\ 1 & 0 & 15 \\ 3 & 3 & 7 \end{pmatrix}$$

We have:

$$C((1, 2, 3, 1)) = d_{12} + d_{23} + d_{31} = 1 + 15 + 3 = 19$$

$$C((1, 3, 2, 1)) = d_{13} + d_{32} + d_{21} = 3 + 3 + 1 = 7$$

Using this framework, the neighbourhood search can be written down as:

1. Start with a given tour: t .
2. Evaluate $C(t)$.
3. Identify a new \tilde{t} from t and accept it as a replacement for t if $C(\tilde{t}) < C(t)$.
4. Repeat the 3rd step until some stopping condition is met.

This is shown diagrammatically in Figure 9.2.

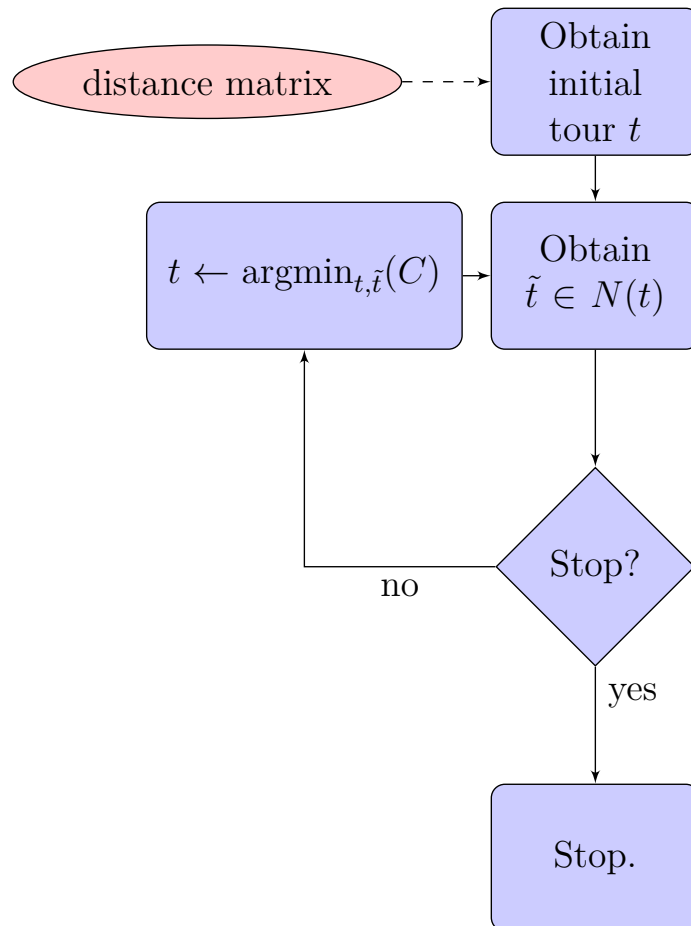


Figure 9.2 The general neighbourhood search algorithm. $N(t)$ refers to some neighbourhood of t .

A number of stopping conditions can be used including some specific overall cost or a number of total iterations of the algorithm.

The neighbourhood of a tour t is taken as some set of tours that can be obtained from t using a specific and computationally efficient **neighbourhood operator**.

To illustrate two such neighbourhoods operators, consider the following tour on 7 stops:

$$t = (0, 1, 2, 3, 4, 5, 6, 0)$$

One possible neighbourhood is to choose 2 stops at random and swap. For example, the tour $\tilde{t}^{(1)} \in N(t)$ is obtained by swapping the 2nd and 5th stops.

$$\tilde{t}^{(1)} = (0, 1, 5, 3, 4, 2, 6, 0)$$

Another possible neighbourhood is to choose 2 stops at random and reversing the order of all stops between (including) those two stops. For example, the tour $\tilde{t}^{(2)} \in N(t)$ is obtained by reversing the order of all stops between the 2nd and the 5th stop.

$$\tilde{t}^{(2)} = (0, 1, 5, 4, 3, 2, 6, 0)$$

Examples of these tours are shown in Figure 9.3.

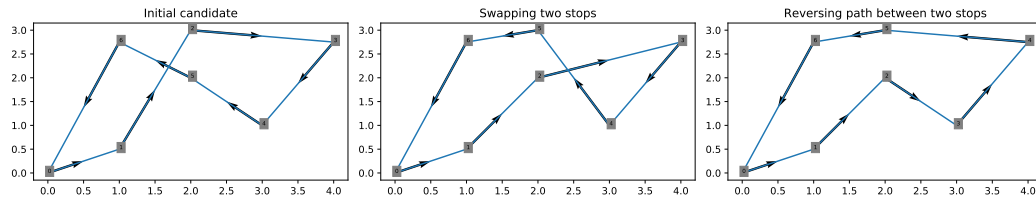


Figure 9.3 The effect of two neighbourhood operators on t . $\tilde{t}^{(1)}$ is obtained by swapping stops 3 and 5. $\tilde{t}^{(2)}$ is obtained by reversing the path between stops 2 and 5.

9.3 SOLVING WITH PYTHON

To solve this problem using Python functions will be written that match the first three steps in the Section 9.2.

The first step is to write the `get_initial_candidate` function that creates an initial tour:

Python input

```

1436 import numpy as np
1437
1438
1439 def get_initial_candidate(number_of_stops, seed):
1440     """Return an random initial tour.
1441
1442     Args:
1443         number_of_stops: The number of stops
1444         seed: An integer seed.
1445
1446     Returns:
1447         A tour starting an ending at stop with index 0.
1448     """
1449     internal_stops = list(range(1, number_of_stops))
1450     np.random.seed(seed)
1451     np.random.shuffle(internal_stops)
1452     return [0] + internal_stops + [0]

```

This gives a random tour on 13 stops:

Python input

```

1453 number_of_stops = 13
1454 seed = 0
1455 initial_candidate = get_initial_candidate(
1456     number_of_stops=number_of_stops,
1457     seed=seed,
1458 )
1459 print(initial_candidate)

```

Python output

```

1460 [0, 7, 12, 5, 11, 3, 9, 2, 8, 10, 4, 1, 6, 0]

```

To be able to evaluate any given tour its cost must be found. Here `get_cost` does this:

Python input

```
1461 def get_cost(tour, distance_matrix):
1462     """Return the cost of a tour.
1463
1464     Args:
1465         tour: A given tuple of successive stops.
1466         distance_matrix: The distance matrix of the problem.
1467
1468     Returns:
1469         The cost
1470     """
1471     return sum(
1472         distance_matrix[current_stop, next_stop]
1473         for current_stop, next_stop in zip(tour[:-1], tour[1:])
1474     )
```

Python input

```

1475 distance_matrix = np.array(
1476     (
1477         (0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1478         (35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1479         (35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1480         (29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1481         (70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1482         (35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1483         (42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1484         (27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1485         (24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1486         (44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1487         (58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1488         (71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1489         (69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0),
1490     )
1491 )
1492 cost = get_cost(
1493     tour=initial_candidate,
1494     distance_matrix=distance_matrix,
1495 )
1496 print(cost)

```

Python output

```

1497 827

```

Now a function for neighbourhood operator will now be written, `swap_stops`, that swaps two stops in a given tour.

Python input

```

1498 def swap_stops(tour):
1499     """Return a new tour by swapping two stops.
1500
1501     Args:
1502         tour: A given tuple of successive stops.
1503
1504     Returns:
1505         A tour
1506     """
1507     number_of_stops = len(tour) - 1
1508     i, j = sorted(
1509         np.random.choice(range(1, number_of_stops), 2)
1510     )
1511     new_tour = list(tour)
1512     new_tour[i], new_tour[j] = tour[j], tour[i]
1513     return new_tour

```

Applying this neighbourhood operator to the initial candidate gives:

Python input

```

1514 print(swap_stops(initial_candidate))

```

which swaps the 10th and 12th stops:

Python output

```

1515 [0, 7, 12, 5, 11, 3, 9, 2, 8, 1, 4, 10, 6, 0]

```

Now all the tools are in place to build a tool to carry out the neighbourhood search `run_neighbourhood_search`.

Python input

```

1516 def run_neighbourhood_search(
1517     distance_matrix,
1518     iterations,
1519     seed,
1520     neighbourhood_operator=swap_stops,
1521 ):
1522     """Returns a tour by carrying out a neighbourhood search.
1523
1524     Args:
1525         distance_matrix: the distance matrix
1526         iterations: the number of iterations for which to
1527             run the algorithm
1528         seed: a random seed
1529         neighbourhood_operator: the neighbourhood operator
1530             (default: swap_stops)
1531
1532     Returns:
1533         A tour
1534     """
1535     number_of_stops = len(distance_matrix)
1536     candidate = get_initial_candidate(
1537         number_of_stops=number_of_stops,
1538         seed=seed,
1539     )
1540
1541     best_cost = get_cost(
1542         tour=candidate,
1543         distance_matrix=distance_matrix,
1544     )
1545
1546     for _ in range(iterations):
1547         new_candidate = neighbourhood_operator(candidate)
1548         if (
1549             cost := get_cost(
1550                 tour=new_candidate,
1551                 distance_matrix=distance_matrix,
1552             )
1553             <= best_cost:
1554             best_cost = cost
1555             candidate = new_candidate
1556
1557     return candidate

```

Now running this for 1000 iterations:

Python input

```
1558 number_of_iterations = 1000
1559
1560 solution_with_swap_stops = run_neighbourhood_search(
1561     distance_matrix=distance_matrix,
1562     iterations=number_of_iterations,
1563     seed=seed,
1564     neighbourhood_operator=swap_stops,
1565 )
1566 print(solution_with_swap_stops)
```

gives:

Python output

```
1567 [0, 7, 2, 8, 5, 3, 1, 9, 12, 11, 4, 10, 6, 0]
```

This has a cost:

Python input

```
1568 cost = get_cost(
1569     tour=solution_with_swap_stops,
1570     distance_matrix=distance_matrix,
1571 )
1572 print(cost)
```

Python output

```
1573 362
```

Therefore, using this particular algorithm, a pretty good route it found, with a total distance of 362.

It is important to note that this may not be the optimal route, and tweaking the

algorithm slightly may produce better solutions. For example, one way to do this is to use a different neighbourhood operator. One example is instead of swapping two stop, revering the path between those two stops. The `reverse_path` function does this:

Python input

```

1574 def reverse_path(tour):
1575     """Return a new tour by reversing the path between two
1576     stops.
1577
1578     Args:
1579         tour: A given tuple of successive stops.
1580
1581     Returns:
1582         A tour
1583     """
1584     number_of_stops = len(tour) - 1
1585     i, j = sorted(
1586         np.random.choice(range(1, number_of_stops), 2)
1587     )
1588     new_tour = tour[:i] + tour[i : j + 1][::-1] + tour[j + 1 :]
1589     return new_tour

```

Applying this neighbourhood operator to the initial candidate gives:

Python input

```

1590 print(reverse_path(initial_candidate))

```

which reverses the order between the 3rd and the 11th stop:

Python output

```

1591 [0, 7, 4, 10, 8, 2, 9, 3, 11, 5, 12, 1, 6, 0]

```

Now running the neighbourhood search for 1000 iterations using the `reverse_path` neighbourhood operator, which corresponds to an algorithm called the “2 opt” algorithm:

Python input

```

1592 solution_with_reverse_path = run_neighbourhood_search(
1593     distance_matrix=distance_matrix,
1594     iterations=number_of_iterations,
1595     seed=seed,
1596     neighbourhood_operator=reverse_path,
1597 )
1598 print(solution_with_reverse_path)

```

gives:

Python output

```

1599 [0, 8, 5, 3, 1, 9, 12, 11, 4, 10, 6, 2, 7, 0]

```

This is now gives a different route. Importantly, the costs differ substantially:

Python input

```

1600 cost = get_cost(
1601     tour=solution_with_reverse_path,
1602     distance_matrix=distance_matrix,
1603 )
1604 print(cost)

```

which gives:

Python output

```

1605 299

```

an improvement on the solution found using the `swap_stops` operator. Figure 9.4 shows the final obtained routes given by both approaches.

9.4 SOLVING WITH R

To solve this problem using R functions will be written that match the first three steps in the Section 9.2.

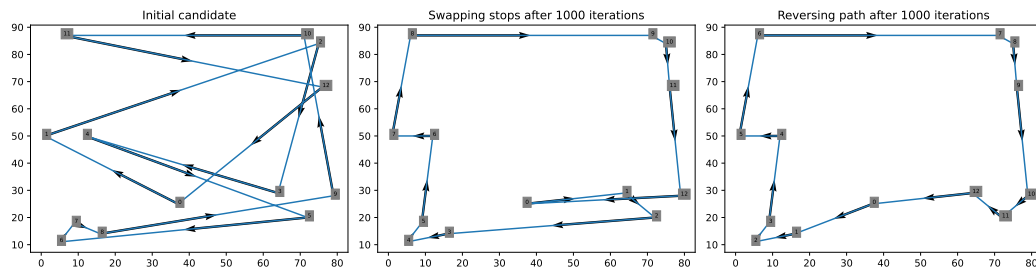


Figure 9.4 The final tours obtained by using the neighbourhood search in Python.

The first step is to write the `get_initial_candidate` function that creates an initial tour:

R input

```

1606 #' Return an random initial tour.
1607 #'
1608 #' @param number_of_stops The number of stops.
1609 #' @param seed An integer seed.
1610 #'
1611 #' @return A tour starting an ending at stop with index 0.
1612 get_initial_candidate <- function(number_of_stops, seed){
1613   internal_stops <- 1:(number_of_stops - 1)
1614   set.seed(seed)
1615   internal_stops <- sample(internal_stops)
1616   c(0, internal_stops, 0)
1617 }

```

This gives a random tour on 13 stops:

R input

```

1618 number_of_stops <- 13
1619 seed <- 1
1620 initial_candidate <- get_initial_candidate(
1621   number_of_stops = number_of_stops,
1622   seed = seed)
1623 print(initial_candidate)

```

R output

1624

```
[1] 0 9 4 7 1 2 5 3 8 6 11 12 10 0
```

To be able to evaluate any given tour its cost must be found. Here `get_cost` to do this:

R input

1625

```
#' Return the cost of a tour
```

1626

```
#'
```

1627

```
#' @param tour A given vector of successive stops.
```

1628

```
#' @param seed The distance matrix of the problem.
```

1629

```
#'
```

1630

```
#' @return The cost
```

1631

```
get_cost <- function(tour, distance_matrix){
```

1632

```
  pairs <- cbind(tour[-length(tour)], tour[-1]) + 1
```

1633

```
  sum(distance_matrix[pairs])
```

1634

```
}
```

R input

```

1635 distance_matrix <- rbind(
1636     c(0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1637     c(35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1638     c(35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1639     c(29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1640     c(70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1641     c(35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1642     c(42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1643     c(27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1644     c(24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1645     c(44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1646     c(58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1647     c(71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1648     c(69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0)
1649 )
1650 cost <- get_cost(
1651     tour = initial_candidate,
1652     distance_matrix = distance_matrix)
1653 print(cost)

```

R output

```

1654 [1] 709

```

Now a function for neighbourhood operator will now be written, `swap_stops`, that swaps two stops in a given tour.

R input

```

1655 #' Return a new tour by swapping two stops.
1656 #'
1657 #' @param tour A given vector of successive stops.
1658 #'
1659 #' @return A tour
1660 swap_stops <- function(tour){
1661   number_of_stops <- length(tour) - 1
1662   stops_to_swap <- sort(sample(2:number_of_stops, 2))
1663   new_tour <- replace(x = tour,
1664                      list = stops_to_swap,
1665                      values = rev(tour[stops_to_swap]))
1666 }

```

Applying this neighbourhood operator to the initial candidate gives:

R input

```

1667 print(swap_stops(initial_candidate))

```

which swaps the 6th and 11th stops:

R output

```

1668 [1] 0 9 4 7 1 11 5 3 8 6 2 12 10 0

```

Now we have all the tools in place to build a tool to carry out the neighbourhood search `run_neighbourhood_search`.

R input

```

1669 #' Returns a tour by carrying out a neighbourhood search
1670 #'
1671 #' @param distance_matrix: the distance matrix
1672 #' @param iterations: the number of iterations for
1673 #'                      which to run the algorithm
1674 #' @param seed: a random seed (default: None)
1675 #' @param neighbourhood_operator: the neighbourhood operation
1676 #'                                (default: swap_stops)
1677 #'
1678 #' @return A tour
1679 run_neighbourhood_search <- function(
1680   distance_matrix,
1681   iterations,
1682   seed = NA,
1683   neighbourhood_operator = swap_stops
1684 ){
1685   number_of_stops <- nrow(distance_matrix)
1686   candidate <- get_initial_candidate(
1687     number_of_stops = number_of_stops,
1688     seed = seed
1689   )
1690
1691   best_cost <- get_cost(
1692     tour = candidate,
1693     distance_matrix = distance_matrix
1694   )
1695
1696   for (repetition in 1:iterations) {
1697     new_candidate <- neighbourhood_operator(candidate)
1698     cost <- get_cost(
1699       tour = new_candidate,
1700       distance_matrix = distance_matrix)
1701
1702     if (cost <= best_cost) {
1703       best_cost <- cost
1704       candidate <- new_candidate
1705     }
1706   }
1707   candidate
1708 }
1709

```

Now running this for 1000 iterations:

R input

```

1710 number_of_iterations <- 1000
1711 solution_with_swap_stops <- run_neighbourhood_search(
1712     distance_matrix = distance_matrix,
1713     iterations = number_of_iterations,
1714     seed = seed,
1715     neighbourhood_operator = swap_stops
1716 )
1717 print(solution_with_swap_stops)

```

gives:

R output

```

1718 [1] 0 11 4 10 6 2 7 12 9 1 3 5 8 0

```

This has a cost:

R input

```

1719 cost <- get_cost(
1720     tour = solution_with_swap_stops,
1721     distance_matrix = distance_matrix
1722 )
1723 print(cost)

```

which gives:

R output

```

1724 [1] 360

```

Therefore, using this particular algorithm, a pretty good route it found, with a total distance of 373.

It is important to note that this may not be the optimal route, and tweaking the

algorithm slightly may produce better solutions. For example, one way to do this is to use a different neighbourhood operator. One example is instead of swapping two stop, revering the path between those two stops. The `reverse_path` function does this:

R input

```

1725 #' Return a new tour by reversing the path between two stops.
1726 #'
1727 #' @param tour A given vector of successive stops.
1728 #'
1729 #' @return A tour
1730 reverse_path <- function(tour){
1731   number_of_stops <- length(tour) - 1
1732   stops_to_swap <- sort(sample(2:number_of_stops, 2))
1733   i <- stops_to_swap[1]
1734   j <- stops_to_swap[2]
1735   new_order <- c(
1736     c(1: (i - 1)),
1737     c(j:i),
1738     c( (j + 1): length(tour))
1739   )
1740   tour[new_order]
1741 }

```

Applying this neighbourhood operator to the initial candidate gives:

R input

```

1742 print(reverse_path(initial_candidate))

```

which reverses the order between the 3rd and the 13th stop:

R output

```

1743 [1] 0 9 10 12 11 6 8 3 5 2 1 7 4 0

```

Now running the neighbourhood search for 1000 iterations using the `reverse_path` neighbourhood operator, which corresponds to an algorithm called the “2 opt” algorithm:

R input

```

1744 number_of_iterations <- 1000
1745 solution_with_reverse_path <- run_neighbourhood_search(
1746     distance_matrix = distance_matrix,
1747     iterations = number_of_iterations,
1748     seed = seed,
1749     neighbourhood_operator = reverse_path
1750 )
1751 print(solution_with_reverse_path)

```

gives:

R output

```

1752 [1] 0 7 2 6 10 4 11 12 9 1 3 5 8 0

```

This is now gives a different route. Importantly, the costs differ substantially:

R input

```

1753 cost <- get_cost(
1754     tour = solution_with_reverse_path,
1755     distance_matrix = distance_matrix
1756 )
1757 print(cost)

```

which gives:

R output

```

1758 [1] 299

```

an improvement on the solution found using the `swap_stops` operator. Figure 9.5 shows the final obtained routes given by both approaches.

9.5 RESEARCH

TBA

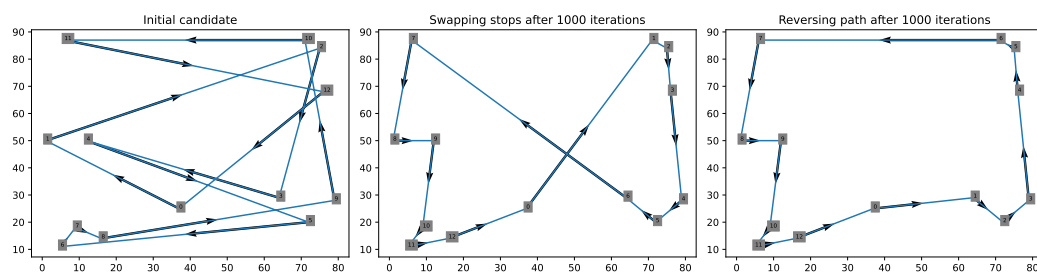


Figure 9.5 The final tours obtained by using the neighbourhood search in R



Bibliography

