
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	36
3.5	RESEARCH HIGHLIGHTS	42

SECTION III Dynamical Systems

CHAPTER	4 ■ Modelling with Differential Equations	45
---------	---	----

4.1	PROBLEM	45
4.2	THEORY	45
4.3	SOLVING WITH PYTHON	46
4.4	SOLVING WITH R	51
4.5	RESEARCH	54

CHAPTER	5 ■ Systems dynamics	55
---------	----------------------	----

5.1	PROBLEM	55
5.2	THEORY	55
5.3	SOLVING WITH PYTHON	58
5.4	SOLVING WITH R	66
5.5	RESEARCH	73

SECTION IV Emergent Behaviour

CHAPTER	6 ■ Game Theory	77
---------	-----------------	----

6.1	PROBLEM	77
6.2	THEORY	78
6.3	SOLVING WITH PYTHON	79
6.4	SOLVING WITH R	83
6.5	RESEARCH	86

CHAPTER	7 ■ Agent Based Simulation	87
---------	----------------------------	----

7.1	PROBLEM	87
7.2	THEORY	87
7.3	SOLVING WITH PYTHON	90
7.4	SOLVING WITH R	95
7.5	RESEARCH	101

SECTION V Optimisation

CHAPTER	8 ■ Linear programming	105
8.1	PROBLEM	105
8.2	THEORY	106
8.3	SOLVING WITH PYTHON	110
8.4	SOLVING WITH R	114
8.5	RESEARCH	122
CHAPTER	9 ■ Heuristics	123
9.1	PROBLEM	123
9.2	THEORY	123
9.3	SOLVING WITH PYTHON	125
9.4	SOLVING WITH R	134
9.5	RESEARCH	142
	Bibliography	145



Foreword

This is the foreword



Preface

This is the preface.



Contributors

Michaél Aftosmis

NASA Ames Research Center
Moffett Field, California

Pratul K. Agarwal

Oak Ridge National Laboratory
Oak Ridge, Tennessee

Sadaf R. Alam

Oak Ridge National Laboratory
Oak Ridge, Tennessee

Gabrielle Allen

Louisiana State University
Baton Rouge, Louisiana

Martin Sandve Alnæs

Simula Research Laboratory and University
of Oslo, Norway
Norway

Steven F. Ashby

Lawrence Livermore National Laboratory
Livermore, California

David A. Bader

Georgia Institute of Technology
Atlanta, Georgia

Benjamin Bergen

Los Alamos National Laboratory
Los Alamos, New Mexico

Jonathan W. Berry

Sandia National Laboratories
Albuquerque, New Mexico

Martin Berzins

University of Utah

Salt Lake City, Utah

Abhinav Bhatele

University of Illinois
Urbana-Champaign, Illinois

Christian Bischof

RWTH Aachen University
Germany

Rupak Biswas

NASA Ames Research Center
Moffett Field, California

Eric Bohm

University of Illinois
Urbana-Champaign, Illinois

James Bordner

University of California, San Diego
San Diego, California

Geörge Bosilca

University of Tennessee
Knoxville, Tennessee

Grèg L. Bryan

Columbia University
New York, New York

Marian Bubak

AGH University of Science and Technology
Kraków, Poland

Andrew Canning

Lawrence Berkeley National Laboratory
Berkeley, California

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 we expect to happen. One tool used to understand randomness is Markov chains, an area of mathematics sitting at the intersection of probability and linear algebra.

2.1 PROBLEM

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

- They currently have 2 barber chairs (and 2 barbers).
- They have waiting room for 4 people.
- They usually have 10 customers arrive per hour.
- Each Barber takes about 15 minutes to serve a customer so they can serve 4 customers an hour.

This is represented diagrammatically in Figure 2.1.

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

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

2.2 THEORY

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

For example, in the barber shop a single number is sufficient to describe the status of the shop. If that number is 1 this implies that 1 customer is currently having their

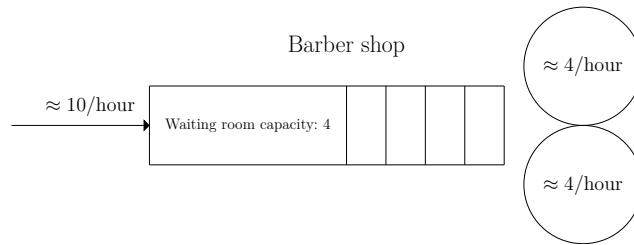


Figure 2.1 Diagrammatic representation of the barber shop as a queuing system.

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

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

As customers arrive and leave the system goes between states as shown in Figure 2.2.

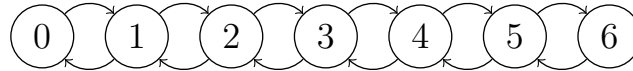


Figure 2.2 Diagrammatic representation of the state space

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

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

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

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

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

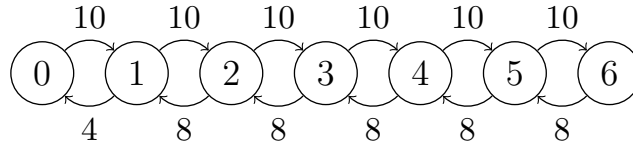


Figure 2.3 Diagrammatic representation of the state space and the transition rates

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

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 .

We can use Q 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 . We can use Q to calculate P_t using the matrix exponential:

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

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

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

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

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

2.3 SOLVING WITH PYTHON

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

Python input

```

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

Python input

```

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 we can obtain the matrix Q for our default system:

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]]

```

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

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 we obtain:

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]]
```

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

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

Python input

```

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

```

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

Python input

```

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

```

we get:

Python output

```

110 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]

```

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

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

Python input

```

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]

```

We 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

```

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

Python input

```

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

```

which gives:

Python output

```
0.23557
```

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

Python input

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

Python output

```
0.078636
```

2.4 SOLVING WITH R

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

R input

```

135 #' Return the transition rate for two 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 }

```

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

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 we can obtain the matrix Q for our default system:

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

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

R input

```

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 we obtain:

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
```

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

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

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 $Ax = b$.

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

R input

```

234 print(get_steady_state_vector(Q))

```

we get:

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

```

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

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

R input

```

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
252    arrival_rate <- 10
253    service_rate <- 4
254    pi <- get_transition_rate_matrix(
255      waiting_room = waiting_room,
256      num_barbers = num_barbers
257    ) %>%
258      get_steady_state_vector()
259
260    capacity <- waiting_room + num_barbers
261    pi[capacity + 1]
262  }

```

Now we can run this code efficiently with both scenarios:

R input

```

262  print(get_probability_of_full_shop(waiting_room = 6))

```

which decreases the probability of a full shop to:

R output

```

263  [1] 0.2355699

```

but adding another barber and chair:

R input

264

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

gives:

R output

265

```
[1] 0.0786359
```

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

```

266 import ciw
267
268
269 def build_network_object(
270     num_inspectors=1,
271     num_repairers=2,
272 ):
273     """Returns a Network object that defines the repair shop.
274
275     Args:
276         num_inspectors: a positive integer (default: 1)
277         num_repairers: a positive integer (default: 2)
278
279     Returns:
280         a Ciw network object
281     """
282     arrival_rate = 15
283     inspection_rate = 20
284     repair_rate = 10
285     prob_need_repair = 0.8
286     N = ciw.create_network(
287         arrival_distributions=[
288             ciw.dists.Exponential(arrival_rate),
289             ciw.dists.NoArrivals(),
290         ],
291         service_distributions=[
292             ciw.dists.Exponential(inspection_rate),
293             ciw.dists.Exponential(repair_rate),
294         ],
295         number_of_servers=[num_inspectors, num_repairers],
296         routing=[[0.0, prob_need_repair], [0.0, 0.0]],
297     )
298     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

```

299 N = build_network_object()
300 print(N.number_of_nodes)

```

which gives:

Python output

```

301 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

```

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

```

317 import pandas as pd
318
319
320 def get_proportion(Q):
321     """Returns the proportion of bicycles spending over a given
322     limit at the repair shop.
323
324     Args:
325     Q: a Ciw simulation object after a run of the
326     simulation
327
328     Returns:
329     a real
330     """
331     limit = 0.5
332     inds = Q.nodes[-1].all_individuals
333     recs = pd.DataFrame(
334         [dr for ind in inds for dr in ind.data_records]
335     )
336     recs["total_time"] = (
337         recs["exit_date"] - recs["arrival_date"]
338     )
339     total_times = recs.groupby("id_number")["total_time"].sum()
340     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

```

341 N = build_network_object()
342 Q = run_simulation(N)
343 p = get_proportion(Q)
344 print(round(p, 6))

```

This gives:

Python output

345 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

```

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

```
368 p = get_average_proportion(num_inspectors=1, num_repairers=2)
369 print(round(p, 6))
```

which gives:

Python output

```
370 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

```
371 p = get_average_proportion(num_inspectors=2, num_repairers=2)
372 print(round(p, 6))
```

which gives:

Python output

```
373 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

```
374 p = get_average_proportion(num_inspectors=1, num_repairers=3)
375 print(round(p, 6))
```

which gives:

Python output

```
376 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.

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 that a bicycle would take at the repair shop:

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



Figure 3.2 Diagrammatic representation of the forked trajectories a bicycle can take

R input

```

377 library(simmer)
378
379 #' Returns a simmer trajectory object outlining the bicycles
380 #' path through the repair shop
381 #'
382 #' @return A simmer trajectory object
383 define_bicycle_trajectories <- function() {
384   inspection_rate <- 20
385   repair_rate <- 10
386   prob_need_repair <- 0.8
387   bicycle <-
388     trajectory("Inspection") %>%
389     seize("Inspector") %>%
390     timeout(function() {
391       rexp(1, inspection_rate)
392     }) %>%
393     release("Inspector") %>%
394     branch(
395       function() (runif(1) < prob_need_repair),
396       continue = c(F),
397       trajectory("Repair") %>%
398         seize("Repairer") %>%
399         timeout(function() {
400           rexp(1, repair_rate)
401         }) %>%
402         release("Repairer"),
403       trajectory("Out")
404     )
405   return(bicycle)
406 }
```

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

```

407 #' Runs one trial of the simulation.
408 #'
409 #' @param bicycle a simmer trajectory object
410 #' @param num_inspectors positive integer (default: 1)
411 #' @param num_repairers positive integer (default: 2)
412 #' @param seed a float (default: 0)
413 #'
414 #' @return A simmer simulation object after one run of
415 #' the simulation
416 run_simulation <- function(bicycle,
417                             num_inspectors = 1,
418                             num_repairers = 2,
419                             seed = 0) {
420   arrival_rate <- 15
421   max_time <- 8
422   repair_shop <-
423     simmer("Repair Shop") %>%
424     add_resource("Inspector", num_inspectors) %>%
425     add_resource("Repairer", num_repairers) %>%
426     add_generator("Bicycle", bicycle, function() {
427       rexp(1, arrival_rate)
428     })
429
430   set.seed(seed)
431   repair_shop %>% run(until = 8)
432   return(repair_shop)
433 }

```

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

```

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

```

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

```

446  bicycle <- define_bicycle_trajectories()
447  repair_shop <- run_simulation(bicycle = bicycle)
448  print(get_proportion(repair_shop = repair_shop))

```

This piece of code gives

R output

```

449  [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

```

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

```

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

R input

```

476  print(
477    get_average_proportion(
478      num_inspectors = 1,
479      num_repairers = 2)
480  )

```

which gives:

R output

```
481 [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

```
482 print(  
483   get_average_proportion(  
484     num_inspectors = 2,  
485     num_repairers = 2)  
486   )
```

which gives:

R output

```
487 [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

```
488 print(  
489   get_average_proportion(  
490     num_inspectors = 1,  
491     num_repairers = 3)  
492   )
```

which gives:

R output

```
493 [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. In this chapter we will consider a direct solution approach using symbolic mathematics.

4.1 PROBLEM

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

4.2 THEORY

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

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

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

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

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

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:

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

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

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

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

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

where K is the cost per person per unit time.

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

4.3 SOLVING WITH PYTHON

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

Python input

```

494 import sympy as sym
495
496 t = sym.Symbol("t")
497 alpha = sym.Symbol("alpha")
498 I_0 = sym.Symbol("I_0")
499 I = sym.Function("I")
500
501
502 def get_equation(alpha=alpha):
503     """Return the differential equation.
504
505     Args:
506         alpha: a float (default: symbolic alpha)
507
508     Returns:
509         A symbolic equation
510     """
511     return sym.Eq(sym.Derivative(I(t), t), -alpha * I(t))

```

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

Python input

```

512 eq = get_equation()
513 print(eq)

```

which gives:

Python output

```

514 Eq(Derivative(I(t), t), -alpha*I(t))

```

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

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

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

Python input

```

515 eq = get_equation(alpha=1)
516 print(eq)

```

Python output

```

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

```

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

Python input

```

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

```

We can verify the solution discussed previously:

Python input

```

530 sol = get_solution()
531 print(sol)

```

which gives:

Python output

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

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

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

Python input

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

which gives:

Python output

```
534 True
```

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

Python input

```
535 sol = get_solution(alpha=2, I_0=100)
536 print(sol)
```

which gives:

Python output

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

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

Python input

```

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

```

We can now obtain the cost without purchasing the cure:

Python input

```

561 I_0 = 100
562 alpha = 2
563 cost_without_cure = get_cost(I_0=I_0, alpha=alpha)
564 print(cost_without_cure)

```

which gives:

Python output

```

565 500

```

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

Python input

```

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

```

which gives:

Python output

```

571 750

```

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, we will actually solve the problem with R using a numerical integration approach. For an outline of the theory behind this approach see Chapter

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

R input

```

572 derivative <- function(t, y, parameters) {
573     with(as.list(c(y, parameters)), {
574         dIdt <- -alpha * I # nolint
575         list(dIdt) # nolint
576     })
577 }

```

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

R input

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

This gives:

R output

```
579 [[1]]
580 [1] -200
```

We will now make use of the `deSolve` library for solving differential equations numerically:

R input

```
581 library(deSolve) # nolint
582 integrate_ode <- function(times,
583                             y0 = c(I = 100),
584                             alpha = 2) {
585   params <- c(alpha = alpha)
586   ode(y = y0, times = times, func = derivative, parms = params)
587 }
```

This will return a sequence of time point and values of I at those time points. Using this we can compute the cost. Note that this function uses `stopifnot` to make sure our differential equation has been solved for a long enough time period.

R input

```

588 get_cost <- function(
589     I_0 = 100,
590     alpha = 2,
591     cost_per_person = 10,
592     cost_of_cure = 0,
593     step_size = 0.0001,
594     max_time = 10) {
595   times <- seq(0, max_time, by = step_size)
596   out <- integrate_ode(times,
597     y0 = c(I = I_0),
598     alpha = alpha
599   )
600   number_of_observations <- length(out[, "I"])
601
602   stopifnot(out[number_of_observations, "I"] < step_size)
603
604   time_between_steps <- diff(out[, "time"])
605   area_under_curve <- sum(
606     time_between_steps *
607     out[-number_of_observations, "I"]
608   )
609   area_under_curve *
610     cost_per_person + cost_of_cure *
611     I_0
612 }

```

Using this we can compute the costs:

R input

```

613 alpha <- 2
614 cost_without_cure <- get_cost(alpha = alpha)
615 print(round(cost_without_cure))

```

which gives:

R output

```
616 [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

```
617 cost_of_cure <- 5
618 cost_with_cure <- get_cost(
619     alpha = 2 * alpha, cost_of_cure = cost_of_cure
620 )
621 print(round(cost_with_cure))
```

which gives:

R output

```
622 [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.

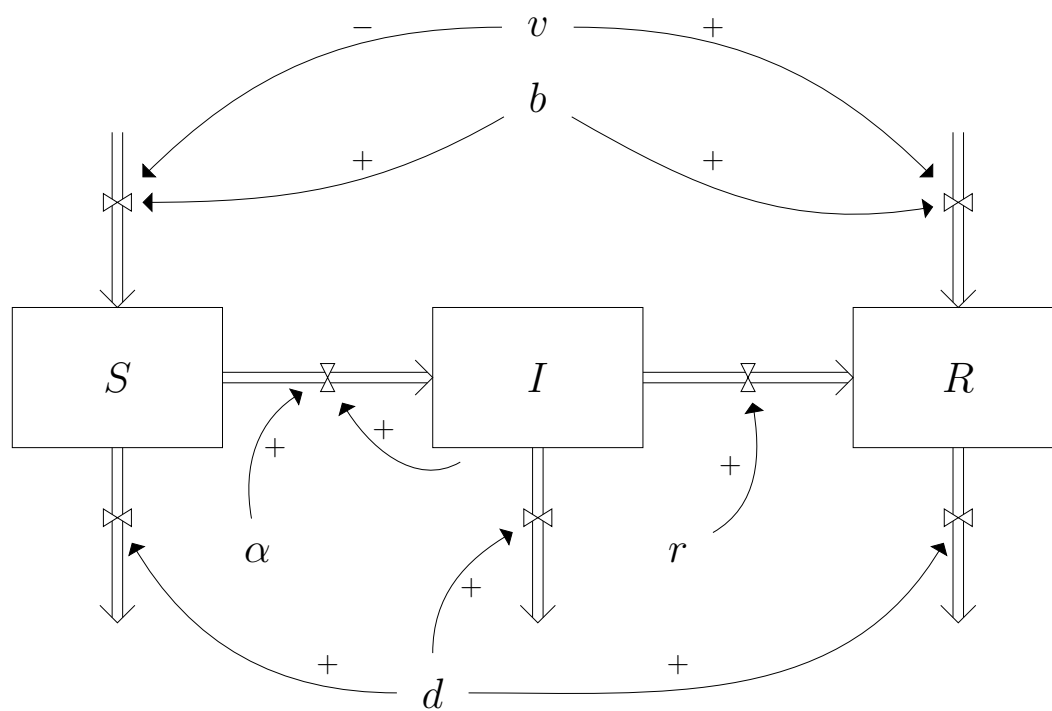


Figure 5.1 Diagrammatic representation of the epidemiology model

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

```

623 def derivatives(y, t, vaccine_rate, birth_rate=0.01):
624     """Defines the system of differential equations that
625     describe the epidemiology model.
626
627     Args:
628         y: a tuple of three integers
629         t: a positive float
630         vaccine_rate: a positive float <= 1
631         birth_rate: a positive float <= 1
632
633     Returns:
634         A tuple containing dS, dI, and dR
635     """
636     infection_rate = 0.3
637     recovery_rate = 0.02
638     death_rate = 0.01
639     S, I, R = y
640     N = S + I + R
641     dSdt = (
642         -((infection_rate * S * I) / N)
643         + ((1 - vaccine_rate) * birth_rate * N)
644         - (death_rate * S)
645     )
646     dIdt = (
647         ((infection_rate * S * I) / N)
648         - (recovery_rate * I)
649         - (death_rate * I)
650     )
651     dRdt = (
652         (recovery_rate * I)
653         - (death_rate * R)
654         + (vaccine_rate * birth_rate * N)
655     )
656     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

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

Python output

```
658 (-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

```

659 from scipy.integrate import odeint
660
661
662 def integrate_ode(
663     derivative_function,
664     t,
665     y0=(2999, 1, 0),
666     vaccine_rate=0.85,
667     birth_rate=0.01,
668 ):
669     """Numerically solve the system of differential equations.
670
671     Args:
672         derivative_function: a function returning a tuple
673             of three floats
674         t: an array of increasing positive floats
675         y0: a tuple of three integers (default: (2999, 1, 0))
676         vaccine_rate: a positive float <= 1 (default: 0.85)
677         birth_rate: a positive float <= 1 (default: 0.01)
678
679     Returns:
680         A tuple of three arrays
681     """
682     results = odeint(
683         derivative_function,
684         y0,
685         t,
686         args=(vaccine_rate, birth_rate),
687     )
688     S, I, R = results.T
689     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%:

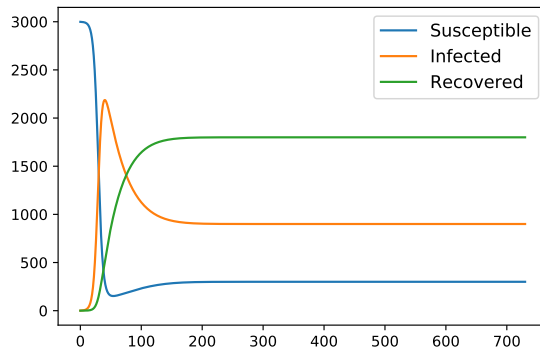


Figure 5.2 Output of code line 737-742

Python input

```

690 import numpy as np
691 from scipy.integrate import odeint
692
693 t = np.arange(0, 730.01, 0.01)
694 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

```

695 import matplotlib.pyplot as plt
696
697 fig, ax = plt.subplots(1)
698 ax.plot(t, S, label='Susceptible')
699 ax.plot(t, I, label='Infected')
700 ax.plot(t, R, label='Recovered')
701 ax.legend(fontsize=12)
702 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

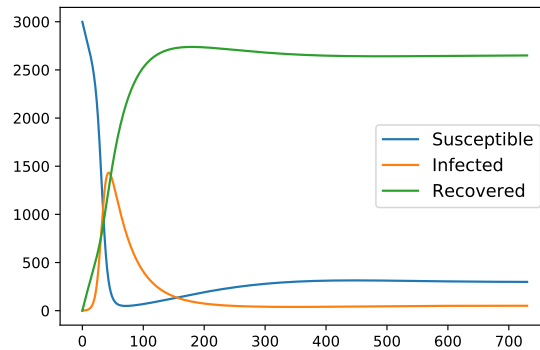


Figure 5.3 Output of code line 745-750

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

```
703 t = np.arange(0, 730.01, 0.01)
704 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

```
705 fig, ax = plt.subplots(1)
706 ax.plot(t, S, label='Susceptible')
707 ax.plot(t, I, label='Infected')
708 ax.plot(t, R, label='Recovered')
709 ax.legend(fontsize=12)
710 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

```

711 def daily_cost(
712     derivative_function=derivatives, vaccine_rate=0.85
713 ):
714     """Calculates the daily cost to the public health system
715     after 2 years.
716
717     Args:
718         derivative_function: a function returning a tuple
719             of three floats
720         vaccine_rate: a positive float <= 1 (default: 0.85)
721
722     Returns:
723         the daily cost
724     """
725     max_time = 730
726     time_step = 0.01
727     birth_rate = 0.01
728     vaccine_cost = 220
729     medication_cost = 10
730     t = np.arange(0, max_time + time_step, time_step)
731     S, I, R = integrate_ode(
732         derivatives,
733         t,
734         vaccine_rate=vaccine_rate,
735         birth_rate=birth_rate,
736     )
737     N = S[-1] + I[-1] + R[-1]
738     daily_vaccine_cost = (
739         N * birth_rate * vaccine_rate * vaccine_cost
740     ) / time_step
741     daily_meds_cost = (I[-1] * medication_cost) / time_step
742     return daily_vaccine_cost + daily_meds_cost

```

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

Python input

```

743 cost = daily_cost(vaccine_rate=0.0)
744 print(round(cost, 2))

```

which gives

Python output

```

745 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

```

746 cost = daily_cost(vaccine_rate=0.85)
747 print(round(cost, 2))

```

which gives

Python output

```

748 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

```

749 #' Defines the system of differential equations that describe
750 #' the epidemiology model.
751 #'
752 #' @param t a positive float
753 #' @param y a tuple of three integers
754 #' @param vaccine_rate a positive float <= 1
755 #' @param birth_rate a positive float <= 1
756 #'
757 #' @return a list containing dS, dI, and dR
758 derivatives <- function(t, y, parameters){
759   infection_rate <- 0.3
760   recovery_rate <- 0.02
761   death_rate <- 0.01
762   with(as.list(c(y, parameters)), {
763     N <- S + I + R
764     dSdt <- ( - ( (infection_rate * S * I) / N) # nolint
765               + ( (1 - vaccine_rate) * birth_rate * N)
766               - (death_rate * S))
767     dIdt <- ( ( (infection_rate * S * I) / N) # nolint
768               - (recovery_rate * I)
769               - (death_rate * I))
770     dRdt <- ( (recovery_rate * I) # nolint
771               - (death_rate * R)
772               + (vaccine_rate * birth_rate * N))
773     list(c(dSdt, dIdt, dRdt)) # nolint
774   })
775 }

```

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

```
776 derivatives(t = 0,  
777           y = c(S = 4, I = 1, R = 0),  
778           parameters = c(vaccine_rate = 0.5,  
779                          birth_rate = 0.01)  
780 )
```

R output

```
781 [[1]]  
782 [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

```

783 library(deSolve) # nolint
784
785 #' Numerically solve the system of differential equations
786 #'
787 #' @param t an array of increasing positive floats
788 #' @param y0 list of integers (default: c(S=2999, I=1, R=0))
789 #' @param birth_rate a positive float <= 1 (default: 0.01)
790 #' @param vaccine_rate a positive float <= 1 (default: 0.85)
791 #'
792 #' @return a matrix of times, S, I and R values
793 integrate_ode <- function(times,
794                             y0 = c(S = 2999, I = 1, R = 0),
795                             birth_rate = 0.01,
796                             vaccine_rate = 0.84){
797   params <- c(birth_rate = birth_rate,
798               vaccine_rate = vaccine_rate)
799   ode(y = y0,
800       times = times,
801       func = derivatives,
802       parms = params)
803 }

```

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

```

804 times <- seq(0, 730, by = 0.01)
805 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.

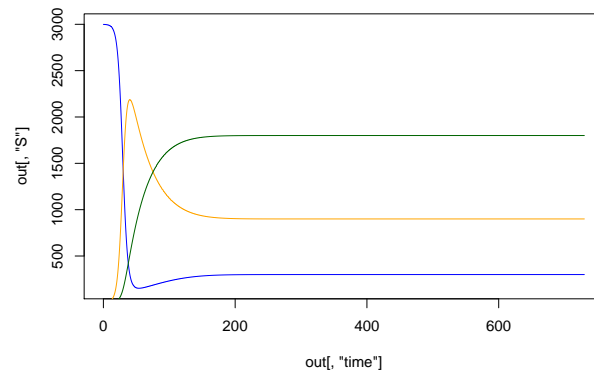


Figure 5.4 Output of code line 846-850

R input

```

806 pdf("plot_no_vaccine_R.pdf", width = 7, height = 5)
807 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
808 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
809 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
810 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

```

811 times <- seq(0, 730, by = 0.01)
812 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.

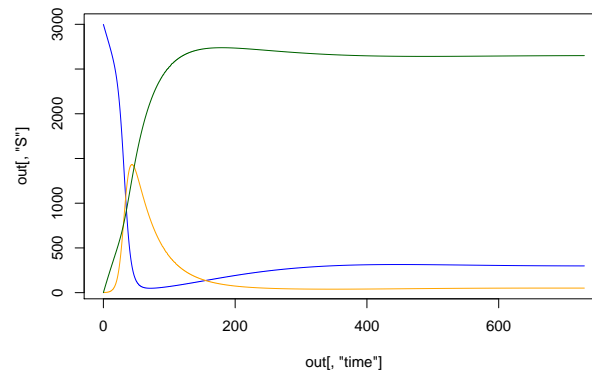


Figure 5.5 Output of code line 853-857

R input

```

813 pdf("plot_with_vaccine_R.pdf", width = 7, height = 5)
814 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
815 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
816 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
817 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

```

818 #' Calculates the daily cost to the public health
819 #' system after 2 years
820 #'
821 #' @param derivative_function: a function returning a
822 #'                               list of three floats
823 #' @param vaccine_rate: a positive float <= 1 (default: 0.85)
824 #'
825 #' @return the daily cost
826 daily_cost <- function(derivative_function = derivatives,
827                        vaccine_rate = 0.85){
828     max_time <- 730
829     time_step <- 0.01
830     birth_rate <- 0.01
831     vaccine_cost <- 220
832     medication_cost <- 10
833     times <- seq(0, max_time, by = time_step)
834     out <- integrate_ode(times, vaccine_rate = vaccine_rate)
835     N <- sum(tail(out[, c("S", "I", "R")], n = 1))
836     daily_vaccine_cost <- (N
837                           * birth_rate
838                           * vaccine_rate
839                           * vaccine_cost) / time_step
840     daily_medication_cost <- ( (tail(out[, "I"], n = 1)
841                               * medication_cost)) / time_step
842     daily_vaccine_cost + daily_medication_cost
843 }

```

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

R input

```

844 cost <- daily_cost(vaccine_rate = 0.0)
845 print(cost)

```

which gives

R output

846

```
[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

847

```
cost <- daily_cost(vaccine_rate = 0.85)
```

848

```
print(cost)
```

which gives

R output

849

```
[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.

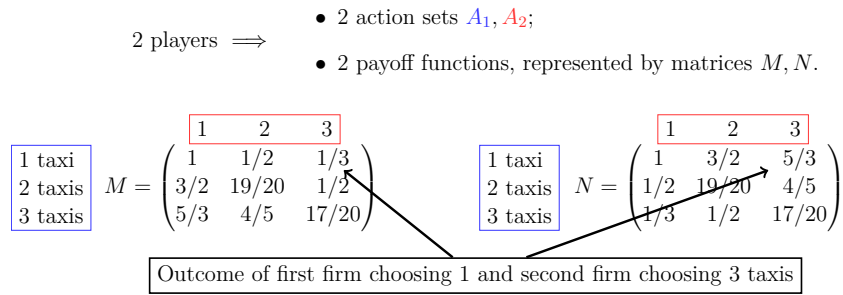


Figure 6.1 Diagrammatic representation of the action sets and payoff matrices for the game.

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

```

850 import nashpy as nash
851
852
853 def get_game(profits):
854     """Return the game object.
855
856     Args:
857         profits: a matrix with expected profits
858
859     Returns:
860         A nashpy game object
861     """
862     return nash.Game(profits, profits.T)

```

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

Python input

```

863 import numpy as np
864
865 profits = np.array(
866     (
867         (1, 1 / 2, 1 / 3),
868         (3 / 2, 19 / 20, 1 / 2),
869         (5 / 3, 4 / 5, 17 / 20),
870     )
871 )
872 game = get_game(profits=profits)
873 print(game)

```

which gives:

Python output

```

874 Bi matrix game with payoff matrices:
875
876 Row player:
877 [[1.          0.5          0.33333333]
878  [1.5         0.95         0.5         ]
879  [1.66666667 0.8          0.85         ]]
880
881 Column player:
882 [[1.          1.5          1.66666667]
883  [0.5         0.95         0.8         ]
884  [0.33333333 0.5          0.85         ]]

```

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

Python input

```

885 for eq in game.support_enumeration():
886     print(eq)

```

which gives:

Python output

```

887 (array([0., 1., 0.]), array([0., 1., 0.]))
888 (array([0., 0., 1.]), array([0., 0., 1.]))
889 (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

```

890 def get_game(profits, offset):
891     """Return the game object with a given offset when 3 taxis
892     are provided.
893
894     Args:
895         profits: a matrix with expected profits
896         offset: a float
897
898     Returns:
899         A nashpy game object
900     """
901     new_profits = np.array(profits)
902     new_profits[2] += offset
903     return nash.Game(new_profits, new_profits.T)

```

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

Python input

```

904 def get_equilibria(profits, offset):
905     """Return the equilibria for a given offset when 3 taxis
906     are provided.
907
908     Args:
909         profits: a matrix with expected profits
910         offset: a float
911
912     Returns:
913         A nashpy game object
914     """
915     game = get_game(profits=profits, offset=offset)
916     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

```

917 offset = 0
918 while len(get_equilibria(profits=profits, offset=offset)) > 1:
919     offset += 0.01

```

This gives a final offset value of:

Python input

```

920 print(round(offset, 2))

```

Python output

```

921 0.15

```

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

Python input

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

giving:

Python output

```
923 ((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

```
924 library(Recon)
925
926 #' Returns the equilibria in pure strategies
927 #'
928 #' @param profits: a matrix with expected profits
929 #'
930 #' @return a list of equilibria
931 get_equilibria <- function(profits){
932     sim_nasheq(profits, t(profits))
933 }
```

Using this we can obtain the pure Nash equilibria:

R input

```

934 profits <- rbind(
935     c(1, 1 / 2, 1 / 3),
936     c(3 / 2, 19 / 20, 1 / 2),
937     c(5 / 3, 4 / 5, 17 / 20)
938 )
939 eqs <- get_equilibria(profits = profits)
940 print(eqs)

```

which gives:

R output

```

941 $`Equilibrium 1`
942 [1] "2" "2"
943
944 $`Equilibrium 2`
945 [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

```

946 #' Returns the equilibria in pure strategies
947 #' for a given offset
948 #'
949 #' @param profits: a matrix with expected profits
950 #' @param offset: a float
951 #'
952 #' @return a list of equilibria
953 get_equilibria <- function(profits, offset){
954   new_profits <- rbind(
955     profits[c(1, 2), ],
956     profits[3, ] + offset)
957   sim_nasheq(new_profits, t(new_profits))
958 }

```

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

R input

```

959 offset <- 0
960 while (length(
961   get_equilibria(profits = profits, offset = offset)
962   ) > 1){
963   offset <- offset + 0.01
964 }

```

This gives a final offset value of:

R input

```

965 print(round(offset, 2))

```

R output

```

966 [1] 0.15

```

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

R input

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

giving:

R output

```
968 $`Equilibrium 1`  
969 [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-

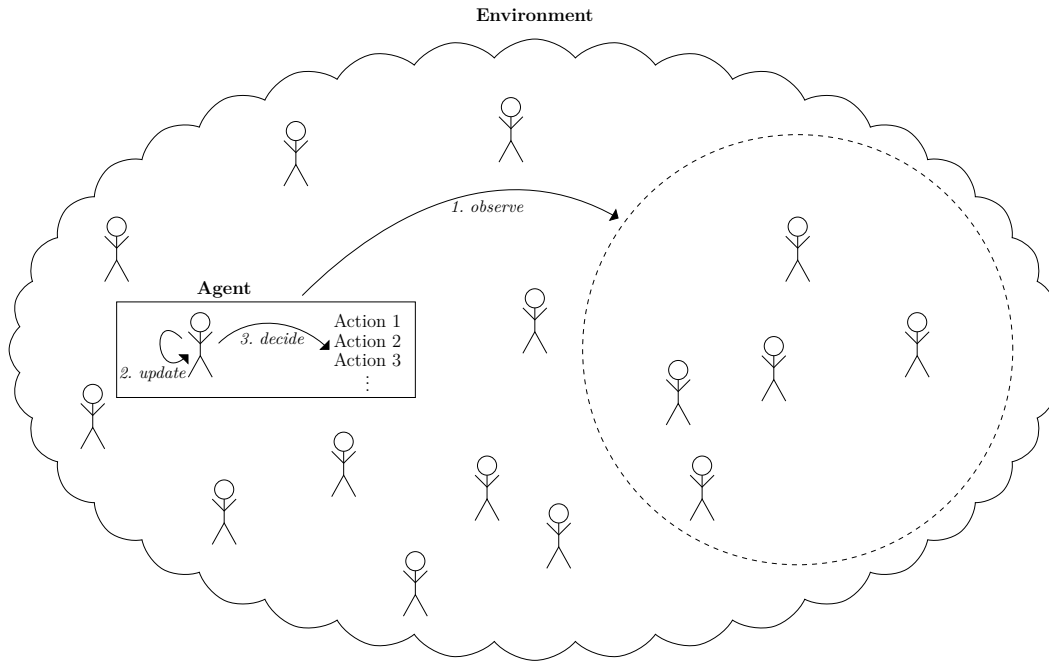


Figure 7.1 Representation of an agent interacting with its environment.

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

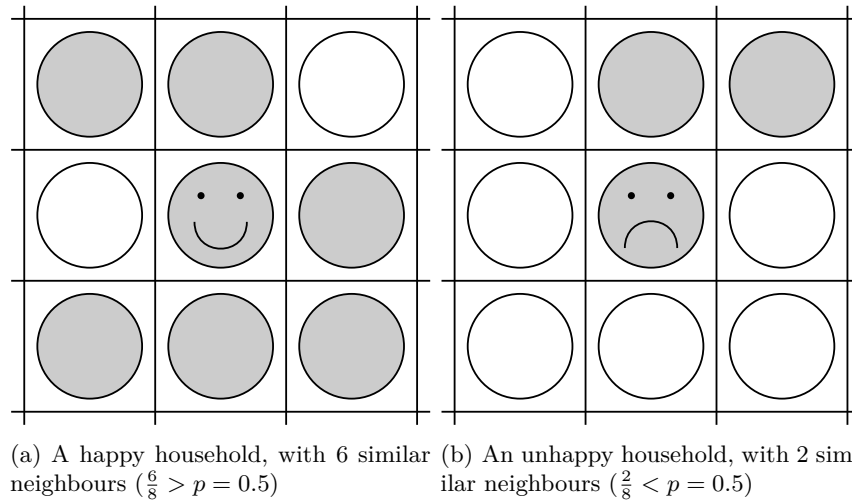


Figure 7.2 Example of a household happy and unhappy with its neighbours, when $p = 0.5$. Households supporting Cardiff City FC are shaded grey, households supporting Swansea City AFC are white.

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
970 import random
971 import itertools
972 import numpy as np
```

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

Python input

```

973 class City:
974     def __init__(self, size, threshold):
975         """Initialises the City object.
976
977         Args:
978             size: an integer number of rows and columns
979             threshold: a number between 0 and 1 representing
980             the minimum acceptable proportion of similar
981             neighbours
982         """
983         self.size = size
984         sides = range(size)
985         self.coords = itertools.product(sides, sides)
986         self.houses = {
987             (x, y): House(x, y, threshold, self)
988             for x, y in self.coords
989         }
990
991     def run(self, n_steps):
992         """Runs the simulation of a number of time steps.
993
994         Args:
995             n_steps: an integer number of steps
996         """
997         for turn in range(n_steps):
998             self.take_turn()
999
1000     def take_turn(self):
1001         """Swaps all sad households."""
1002         sad = [h for h in self.houses.values() if h.sad()]
1003         random.shuffle(sad)
1004         i = 0
1005         while i <= len(sad) / 2:
1006             sad[i].swap(sad[-i])
1007             i += 1
1008
1009     def mean_satisfaction(self):
1010         """Finds the average household satisfaction.
1011
1012         Returns:
1013             The average city's household satisfaction
1014         """
1015         return np.mean(
1016             [h.satisfaction() for h in self.houses.values()]
1017         )

```

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

```

1018 class House:
1019     def __init__(self, x, y, threshold, city):
1020         """Initialises the House object.
1021
1022         Args:
1023             x: the integer x-coordinate
1024             y: the integer y-coordinate
1025             threshold: a number between 0 and 1 representing
1026                 the minimum acceptable proportion of similar
1027                 neighbours
1028             city: an instance of the City class
1029         """
1030         self.x = x
1031         self.y = y
1032         self.threshold = threshold
1033         self.kind = random.choice(["Cardiff", "Swansea"])
1034         self.city = city
1035
1036     def satisfaction(self):
1037         """Determines the household's satisfaction level.
1038
1039         Returns:
1040             A proportion
1041         """
1042         same = 0
1043         for x, y in itertools.product([-1, 0, 1], [-1, 0, 1]):
1044             ax = (self.x + x) % self.city.size
1045             ay = (self.y + y) % self.city.size
1046             same += self.city.houses[ax, ay].kind == self.kind
1047         return (same - 1) / 8
1048
1049     def sad(self):
1050         """Determines if the household is sad.
1051
1052         Returns:
1053             a Boolean
1054         """
1055         return self.satisfaction() < self.threshold
1056
1057     def swap(self, house):
1058         """Swaps two households.
1059
1060         Args:
1061             house: the house object to swap household with
1062         """
1063         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

```

1064 def find_mean_happiness(seed, size, threshold, n_steps):
1065     """Create and run an instance of the simulation.
1066
1067     Args:
1068         seed: the random seed to use
1069         size: an integer number of rows and columns
1070         threshold: a number between 0 and 1 representing
1071             the minimum acceptable proportion of similar
1072             neighbours
1073         n_steps: an integer number of steps
1074
1075     Returns:
1076         The average city's household satisfaction after
1077         n_steps
1078     """
1079     random.seed(seed)
1080     C = City(size, threshold)
1081     C.run(n_steps)
1082     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

```
1083 print(find_mean_happiness(0, 50, 0.65, 0))
```

Python output

```
1084 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

```
1085 print(find_mean_happiness(0, 50, 0.65, 100))
```

Python output

```
1086 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.

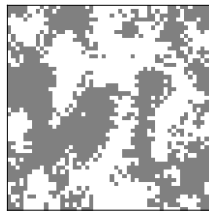
More analysis methods can be added, including plotting functions. Figure 7.3 shows the grid at the beginning, after 20 time steps, and after 100 time steps, with households supporting Cardiff City FC in grey, and those supporting Swansea City AFC in white. It visually shows the households naturally segregating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

7.4 SOLVING WITH R

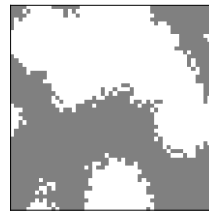
In agent based modelling we consider individual agents as their own entities, with their own rules and behaviours. This world view lends itself well to object-orientated programming. Here we build a number of *objects* from a set of instructions called a



(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

```

1087 library(R6)
1088 city <- R6Class("City", list(
1089   size = NA,
1090   houses = NA,
1091   initialize = function(size, threshold) {
1092     self$size <- size
1093     self$houses <- c()
1094     for (x in 1:size) {
1095       row <- c()
1096       for (y in 1:size) {
1097         row <- c(row, house$new(x, y, threshold, self))
1098       }
1099       self$houses <- rbind(self$houses, row)
1100     } },
1101   run = function(n_steps) {
1102     if (n_steps > 0) {
1103       for (turn in 1:n_steps) {
1104         self$take_turn()
1105       } },
1106   take_turn = function() {
1107     sad <- c()
1108     for (house in self$houses) {
1109       if (house$sad()) {
1110         sad <- c(sad, house)
1111       } }
1112     sad <- sample(sad)
1113     num_sad <- length(sad)
1114     i <- 1
1115     while (i <= num_sad / 2) {
1116       sad[[i]]$swap(sad[[num_sad - i]])
1117       i <- i + 1
1118     } },
1119   mean_satisfaction = function() {
1120     mean(sapply(self$houses, function(x) x$satisfaction()))
1121   })
1122 )

```

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

```

1123 house <- R6Class("House", list(
1124   x = NA,
1125   y = NA,
1126   threshold = NA,
1127   city = NA,
1128   kind = NA,
1129   initialize = function(x = NA,
1130                         y = NA,
1131                         threshold = NA,
1132                         city = NA) {
1133     self$x <- x
1134     self$y <- y
1135     self$threshold <- threshold
1136     self$city <- city
1137     self$kind <- sample(c("Cardiff", "Swansea"), 1)
1138   },
1139   satisfaction = function() {
1140     same <- 0
1141     for (x in -1:1) {
1142       for (y in -1:1) {
1143         ax <- ( (self$x + x - 1) %% self$city$size) + 1
1144         ay <- ( (self$y + y - 1) %% self$city$size) + 1
1145         if (self$city$houses[[ax, ay]]$kind == self$kind) {
1146           same <- same + 1
1147         } } }
1148     (same - 1) / 8
1149   },
1150   sad = function() {
1151     self$satisfaction() < self$threshold
1152   },
1153   swap = function(house) {
1154     old <- self$kind
1155     self$kind <- house$kind
1156     house$kind <- old
1157   })
1158 )

```

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

```

1159  #' Create and run an instance of the simulation.
1160  #'
1161  #' @param seed: the random seed to use
1162  #' @param size: an integer number of rows and columns
1163  #' @param threshold: a number between 0 and 1 representing
1164  #'   the minimum acceptable proportion of similar neighbours
1165  #' @param n_steps: an integer number of steps
1166  #'
1167  #' @return The average city's household satisfaction
1168  #'   after n_steps
1169  find_mean_happiness <- function(seed, size,
1170                                threshold, n_steps){
1171    set.seed(seed)
1172    our_city <- city$new(size, threshold)
1173    our_city$run(n_steps)
1174    our_city$mean_satisfaction()
1175  }

```

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

```

1176  print(find_mean_happiness(0, 50, 0.65, 0))

```

R output

```
1177 [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:

R input

```
1178 print(find_mean_happiness(0, 50, 0.65, 100))
```

R output

```
1179 [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

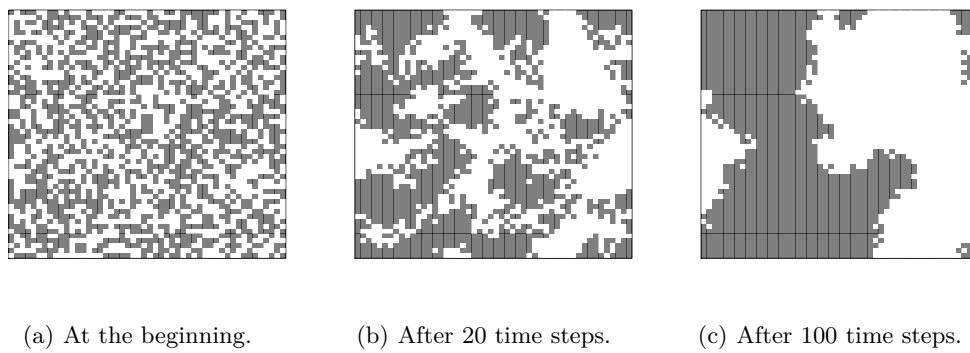


Figure 7.4 Plotted results from the R code.

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:

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

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

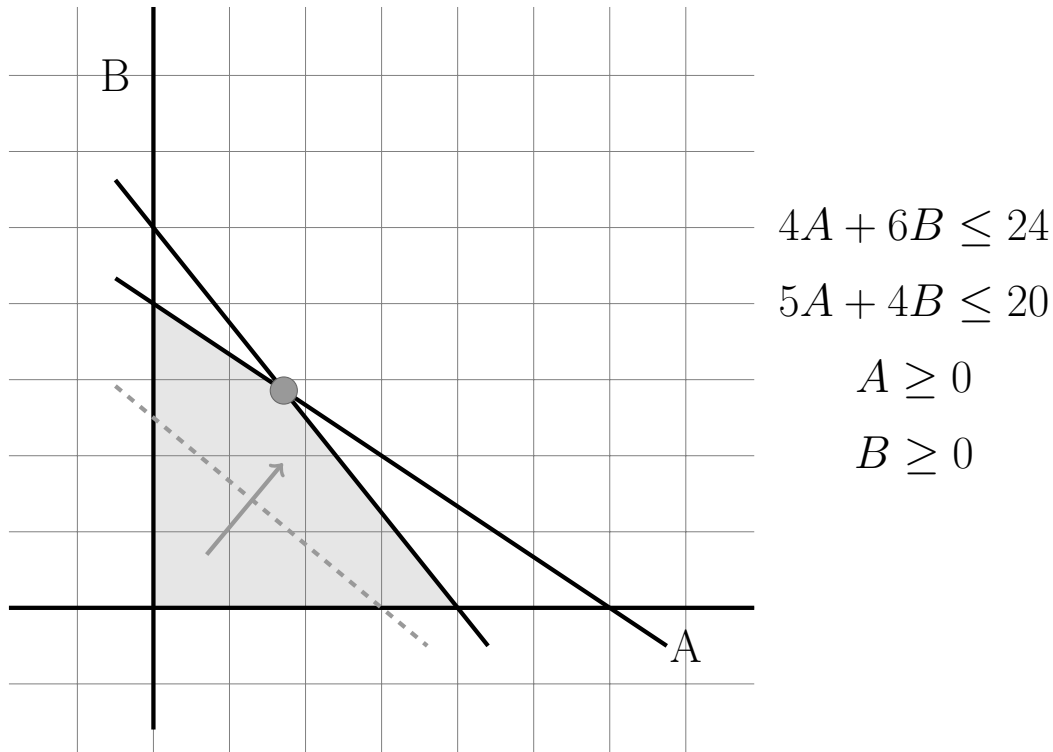


Figure 8.1 Visual representation of the paint linear program. The feasible convex region is shaded in grey; the objective function with arbitrary value is shown in a dashed line.

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

```

1180 Ac = [0, 1]
1181 Ao = [2, 3, 4]
1182 Bc = [5, 6]
1183 Bo = [7, 8]
1184 Cc = [9, 10]
1185 Co = [11, 12, 13]
1186 modules = Ac + Ao + Bc + Bo + Cc + Co
1187 times = range(14)

```

Now let's begin by defining an empty problem:

Python input

```

1188 import pulp
1189
1190 prob = pulp.LpProblem("ExamScheduling", pulp.LpMinimize)

```

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

Python input

```

1191 xshape = (modules, times)
1192 x = pulp.LpVariable.dicts("X", xshape, cat=pulp.LpBinary)
1193 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

```

1194 print(y[3])

```

Python output

1195 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

1196 `print(x[2][5])`

Python output

1197 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

1198 `objective_function = sum([y[day] for day in times])`
 1199 `prob += objective_function`

Now the constraints, Inequalities 8.5-8.10:

Python input

```

1200 M = 1 / len(modules)
1201 for day in times:
1202     prob += M * sum(x[m][day] for m in modules) <= y[day]
1203     prob += sum([x[mod][day] for mod in Ac + Ao]) <= 1
1204     prob += sum([x[mod][day] for mod in Bc + Bo + Co]) <= 1
1205     prob += sum([x[mod][day] for mod in Bc + Bo + Ac]) <= 1
1206     prob += sum([x[mod][day] for mod in Cc + Co + Bo]) <= 1
1207
1208 for mod in modules:
1209     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

```

1210 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

```

1211 print(x[2][5].value())

```

Python output

```

1212 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

```
1213 print(x[2][9].value())
```

Python output

```
1214 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

```
1215 for day in times:
1216     if y[day].value() == 1:
1217         schedule = f"Day {day}: "
1218         for mod in modules:
1219             if x[mod][day].value() == 1:
1220                 schedule += f"{mod}, "
1221         print(schedule)
```

giving:

Python output

```
1222 Day 0: 1, 12,
1223 Day 3: 5, 9,
1224 Day 5: 0, 13,
1225 Day 6: 3, 11,
1226 Day 7: 6, 10,
1227 Day 9: 2, 7,
1228 Day 13: 4, 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 ROI, 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 ROI 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

```

1229  #' Writes the row of coefficients for the objective function
1230  #'
1231  #' @param n_modules: the number of modules to schedule
1232  #' @param n_days: the maximum number of days to schedule
1233  #'
1234  #' @return the objective function row to minimise
1235  write_objective <- function(n_modules, n_days){
1236    all_days <- rep(0, n_modules * n_days)
1237    Ys <- rep(1, n_days)
1238    append(all_days, Ys)
1239  }

```

For 3 modules and 3 days:

R input

```

1240  write_objective(3, 3)

```

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

R output

```

1241  [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

```

1242 #' Writes the constraint row dealing with clashes
1243 #'
1244 #' @param clashes: a vector of module indices that all cannot
1245 #'                  be scheduled at the same time
1246 #' @param day: an integer representing the day
1247 #'
1248 #' @return the constraint row corresponding to that set of
1249 #'         clashes on that day
1250 write_X_clashes <- function(clashes, day, n_days, n_modules){
1251   today <- rep(0, n_modules)
1252   today[clashes] = 1
1253   before_today <- rep(0, n_modules * (day - 1))
1254   after_today <- rep(0, n_modules * (n_days - day))
1255   all_days <- c(before_today, today, after_today)
1256   full_coeffs <- c(all_days, rep(0, n_days))
1257   full_coeffs
1258 }

```

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

```

1259 #' Writes the constraint row to ensure that every module is
1260 #' scheduled once and only one
1261 #'
1262 #' @param module: an integer representing the module
1263 #'
1264 #' @return the constraint row corresponding to scheduling a
1265 #'         module on only one day
1266 write_X_requirements <- function(module, n_days, n_modules){
1267   today <- rep(0, n_modules)
1268   today[module] = 1
1269   all_days <- rep(today, n_days)
1270   full_coeffs <- c(all_days, rep(0, n_days))
1271   full_coeffs
1272 }

```

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

R input

```

1273 #' Writes the constraint row representing the Y variable,
1274 #' whether at least one exam is scheduled on that day
1275 #'
1276 #' @param day: an integer representing the day
1277 #'
1278 #' @return the constraint row corresponding to creating Y
1279 write_Y_constraints <- function(day, n_days, n_modules){
1280   today <- rep(1, n_modules)
1281   before_today <- rep(0, n_modules * (day - 1))
1282   after_today <- rep(0, n_modules * (n_days - day))
1283   all_days <- c(before_today, today, after_today)
1284   all_Ys <- rep(0, n_days)
1285   all_Ys[day] = -n_modules
1286   full_coeffs <- append(all_days, all_Ys)
1287   full_coeffs
1288 }

```

Finally the following function uses them all to assemble a coefficients matrix. It loops through 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

```

1289 #' Writes all the constraints as a matrix, column of
1290 #' inequalities, and right hand side column.
1291 #'
1292 #' @param list_clashes: a list of vectors with sets of modules
1293 #' that cannot be scheduled at the same time
1294 #'
1295 #' @return f.con the LHS of the constraints as a matrix
1296 #' @return f.dir the directions of the inequalities
1297 #' @return f.rhs the values of the RHS of the inequalities
1298 write_constraints <- function(list_clashes, n_days, n_modules){
1299   all_rows <- c()
1300   all_dirs <- c()
1301   all_rhss <- c()
1302   n_rows <- 0
1303
1304   for (clash in list_clashes){
1305     for (day in 1:n_days){
1306       clashes <- write_X_clashes(clash, day, n_days, n_modules)
1307       all_rows <- append(all_rows, clashes)
1308       all_dirs <- append(all_dirs, "<=")
1309       all_rhss <- append(all_rhss, 1)
1310       n_rows <- n_rows + 1
1311     }
1312   }
1313
1314   for (module in 1:n_modules){
1315     reqs <- write_X_requirements(module, n_days, n_modules)
1316     all_rows <- append(all_rows, reqs)
1317     all_dirs <- append(all_dirs, "==")
1318     all_rhss <- append(all_rhss, 1)
1319     n_rows <- n_rows + 1
1320   }
1321
1322   for (day in 1:n_days){
1323     Yconstraints <- write_Y_constraints(day, n_days, n_modules)
1324     all_rows <- append(all_rows, Yconstraints)
1325     all_dirs <- append(all_dirs, "<=")
1326     all_rhss <- append(all_rhss, 0)
1327     n_rows <- n_rows + 1
1328   }
1329
1330   f.con <- matrix(all_rows, nrow = n_rows, byrow = TRUE)
1331   f.dir <- all_dirs
1332   f.rhs <- all_rhss
1333   list(f.con, f.dir, f.rhs)
1334 }

```

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

```
1335 write_constraints(list_clashes = list(c(1, 2)),
1336                   n_days = 2,
1337                   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

```
1338 [[1]]
1339      [,1] [,2] [,3] [,4] [,5] [,6]
1340 [1,]    1    1    0    0    0    0
1341 [2,]    0    0    1    1    0    0
1342 [3,]    1    0    1    0    0    0
1343 [4,]    0    1    0    1    0    0
1344 [5,]    1    1    0    0   -2    0
1345 [6,]    0    0    1    1    0   -2
1346
1347 [[2]]
1348 [1] "<=" "<=" "==" "==" "<=" "<="
1349
1350 [[3]]
1351 [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

```

1352 n_modules = 14
1353 n_days = 14
1354
1355 Ac <- c(0, 1)
1356 Ao <- c(2, 3, 4)
1357 Bc <- c(5, 6)
1358 Bo <- c(7, 8)
1359 Cc <- c(9, 10)
1360 Co <- c(11, 12, 13)
1361
1362 list_clashes <- list(
1363   c(Ac, Ao),
1364   c(Bc, Bo, Co),
1365   c(Bc, Bo, Ac),
1366   c(Bo, Cc, Co)
1367 )

```

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

R input

```

1368 constraints <- write_constraints(list_clashes = list_clashes,
1369                                n_days = n_days,
1370                                n_modules = n_modules)
1371 f.con <- constraints[[1]]
1372 f.dir <- constraints[[2]]
1373 f.rhs <- constraints[[3]]
1374 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

```

1375 library(ROI)
1376
1377 milp <- OP(objective = L_objective(f.obj),
1378           constraints = L_constraint(L = f.con,
1379                                   dir = f.dir,
1380                                   rhs = f.rhs),
1381           types = rep("B", length(f.obj)),
1382           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

```

1383 sol <- ROI_solve(milp)

```

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

R input

```

1384 print(sol$solution)

```

R output

```

1385 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
1386 [30] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1387 [59] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
1388 [88] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
1389 [117] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0
1390 [146] 0 0 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0
1391 [175] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0
1392 [204] 1 0 1 1 1 0 1

```

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

```

1393 for (day in 1:n_days){
1394   if (sol$solution[(n_days * n_modules) + day] == 1){
1395     schedule <- paste("Day", day, ":")
1396     for (module in 1:n_modules){
1397       var <- ((day - 1) * n_modules) + module
1398       if (sol$solution[var] == 1){
1399         schedule <- paste(schedule, module)
1400       }
1401     }
1402     print(schedule)
1403   }
1404 }

```

R output

```

1405 [1] "Day 2 : 4 11"
1406 [1] "Day 6 : 1 12"
1407 [1] "Day 8 : 7"
1408 [1] "Day 10 : 8"
1409 [1] "Day 11 : 3 13"
1410 [1] "Day 12 : 2 6 9 14"
1411 [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 that needs to find itineraries for a driver. In the past, the management team has noticed that drivers will often drive to whichever next stop is closest but this often makes for longer deliveries.

The stops are represented in Figure 9.2.

The distance matrix is given 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, $d_{23} = 89$ indicates that the distance between the 2nd and 3rd stop in the third itinerary is given 89.

Given these parameters, we aim to find a *sufficiently good* set of itineraries that gives a low total amount of travel.

The emphasis on needing a good solution, but not necessarily the best one, prioritising computational efficiency is where the field of heuristics comes in to its own.

9.2 THEORY

The heuristic approach take here will be to use a neighborhood 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

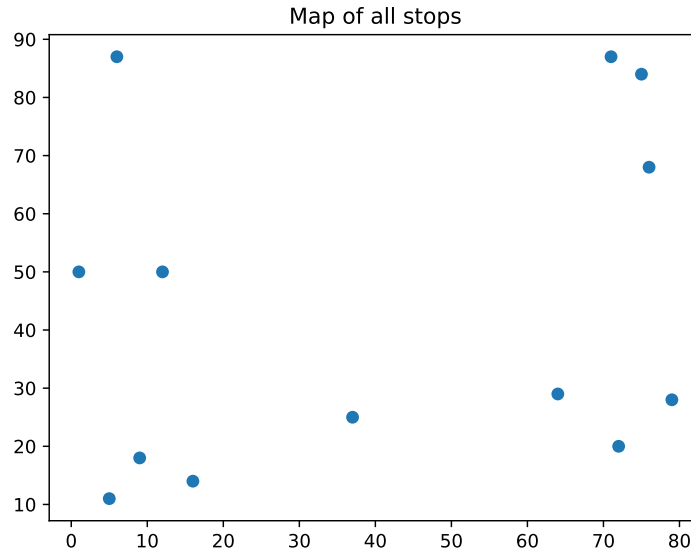


Figure 9.1 Diagrammatic representation of the action sets and payoff matrices for the game.

different approaches and problems: it is referred to as the neighbourhood. As a new solution is evaluated if it is *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 our problem, the first aspect of this is to represent a given trajectory between all the potential stops as a *tour*. If we have 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.

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 $t^{(1)} \in N(t)$ is obtained by swapping the 3rd and 5th stops.

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

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 $t^{(2)} \in N(t)$ is obtained by reversing the order of all stops between the 3rd and the 5th stop.

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

Examples of these tours are shown in Figure 9.3.

9.3 SOLVING WITH PYTHON

To solve this problem using Python we will write functionality that matches 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:

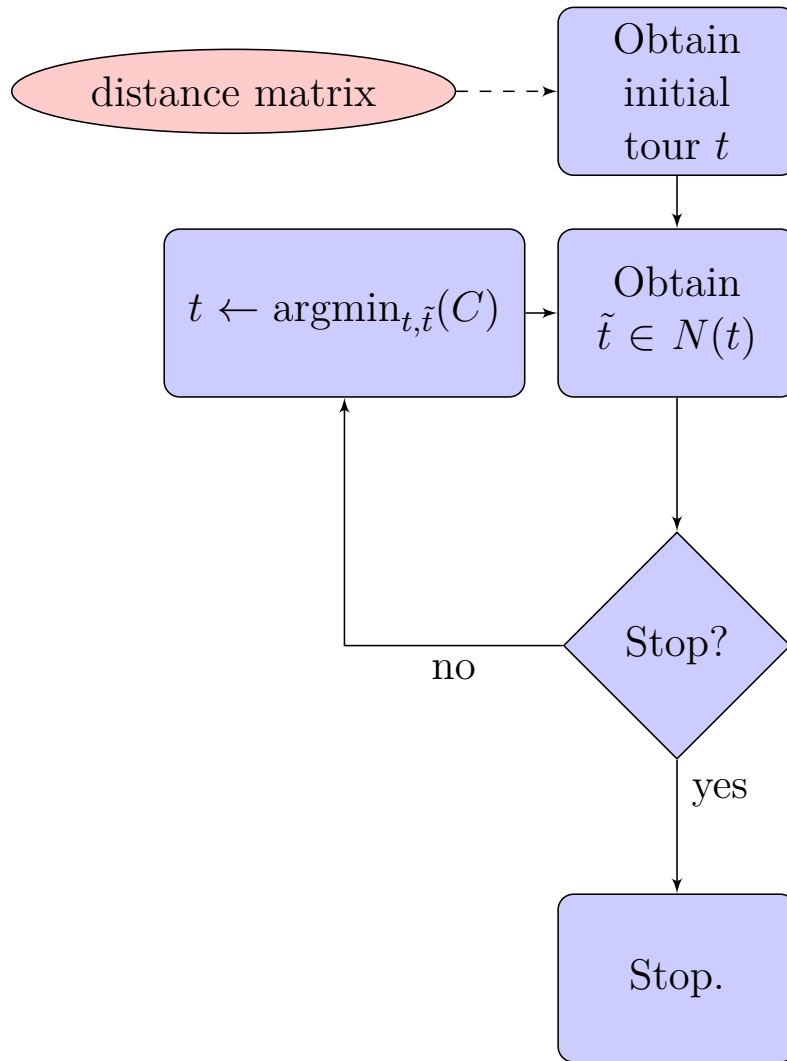


Figure 9.2 The general neighbourhood search algorithm. $N(t)$ refers to some neighbourhood of t .

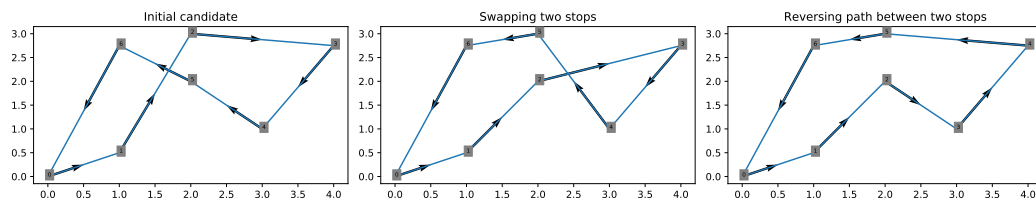


Figure 9.3 The effect of two neighborhood operators on t . $t^{(1)}$ is obtained by swapping stops 3 and 5. $t^{(2)}$ is obtained by reversing the path between stops 3 and 5.

Python input

```

1412 import numpy as np
1413
1414
1415 def get_initial_candidate(number_of_stops, seed=None):
1416     """Return an initial tour.
1417
1418     Args:
1419         number_of_stops: The number of stops
1420         seed: An integer seed. If an integer value is
1421             passed it will create a random tour.
1422
1423     Returns:
1424         A tour starting and ending at stop with index 0.
1425     """
1426     internal_stops = list(range(1, number_of_stops))
1427     if seed is not None:
1428         np.random.seed(seed)
1429         np.random.shuffle(internal_stops)
1430     return [0] + internal_stops + [0]

```

Using this we can get a random tour on 13 stops:

Python input

```

1431 number_of_stops = 13
1432 seed = 0
1433 initial_candidate = get_initial_candidate(
1434     number_of_stops=number_of_stops,
1435     seed=seed,
1436 )
1437 print(initial_candidate)

```

Python output

```

1438 [0, 7, 12, 5, 11, 3, 9, 2, 8, 10, 4, 1, 6, 0]

```

To be able to evaluate any given tour we see that we must also be able to evaluate its cost. Here we define `get_cost` to do this:

Python input

```
1439 def get_cost(tour, distance_matrix):
1440     """Return the cost of a tour.
1441
1442     Args:
1443         tour: A given tuple of successive stops.
1444         distance_matrix: The distance matrix of the problem.
1445
1446     Returns:
1447         The cost
1448     """
1449     return sum(
1450         distance_matrix[current_stop, next_stop]
1451         for current_stop, next_stop in zip(tour[:-1], tour[1:])
1452     )
```


Python input

```

1453 distance_matrix = np.array(
1454     (
1455         (0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1456         (35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1457         (35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1458         (29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1459         (70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1460         (35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1461         (42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1462         (27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1463         (24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1464         (44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1465         (58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1466         (71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1467         (69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0),
1468     )
1469 )
1470 cost = get_cost(
1471     tour=initial_candidate,
1472     distance_matrix=distance_matrix,
1473 )
1474 print(cost)

```

Python output

```

1475 827

```

We will now define two different neighbourhood operators:

- `swap_stops`: this swaps two stops in a given tour.
- `reverse_path`: this swaps two stops and reverts the stops in between them.

Python input

```

1476 def swap_stops(tour):
1477     """Return a new tour by swapping two stops.
1478
1479     Args:
1480         tour: A given tuple of successive stops.
1481
1482     Returns:
1483         A tour
1484     """
1485     number_of_stops = len(tour) - 1
1486     i, j = sorted(
1487         np.random.choice(range(1, number_of_stops), 2)
1488     )
1489     new_tour = list(tour)
1490     new_tour[i], new_tour[j] = tour[j], tour[i]
1491     return new_tour
1492
1493
1494 def reverse_path(tour):
1495     """Return a new tour by reversing the path between two
1496     stops.
1497
1498     Args:
1499         tour: A given tuple of successive stops.
1500
1501     Returns:
1502         A tour
1503     """
1504     number_of_stops = len(tour) - 1
1505     i, j = sorted(
1506         np.random.choice(range(1, number_of_stops), 2)
1507     )
1508     new_tour = tour[:i] + tour[i : j + 1][::-1] + tour[j + 1 :]
1509     return new_tour

```

If we apply these two neighbourhood operators to our initial candidate we can see the effects:

Python input

```
1510 print(swap_stops(initial_candidate))
```

which swaps the 3rd and 8th stops:

Python output

```
1511 [0, 7, 12, 5, 11, 3, 9, 2, 8, 1, 4, 10, 6, 0]
```

Python input

```
1512 print(reverse_path(initial_candidate))
```

which reverses the order between the 3rd and the 8th stop:

Python output

```
1513 [0, 7, 2, 9, 3, 11, 5, 12, 8, 10, 4, 1, 6, 0]
```

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

Python input

```

1514 def run_neighbourhood_search(
1515     distance_matrix,
1516     number_of_stops,
1517     iterations,
1518     seed=None,
1519     neighbourhood_operator=reverse_path,
1520 ):
1521     """Returns a tour by carrying out a neighbourhood search.
1522
1523     Args:
1524         distance_matrix: the distance matrix
1525         number_of_stops: the number of stops
1526         iterations: the number of iterations for which to
1527             run the algorithm
1528         seed: a random seed (default: None)
1529         neighbourhood_operator: the neighbourhood operator
1530             (default: reverse_path)
1531
1532     Returns:
1533         A tour
1534     """
1535     candidate = get_initial_candidate(
1536         number_of_stops=number_of_stops,
1537         seed=seed,
1538     )
1539
1540     best_cost = get_cost(
1541         tour=candidate,
1542         distance_matrix=distance_matrix,
1543     )
1544
1545     for _ in range(iterations):
1546         new_candidate = neighbourhood_operator(candidate)
1547         if (
1548             cost := get_cost(
1549                 tour=new_candidate,
1550                 distance_matrix=distance_matrix,
1551             )
1552         ) <= best_cost:
1553             best_cost = cost
1554             candidate = new_candidate
1555
1556     return candidate

```

Using this we can see the effect of running 1000 iterations using different neighbourhood functions:

Python input

```

1557 number_of_iterations = 1000
1558
1559 solution_with_swap_stops = run_neighbourhood_search(
1560     distance_matrix=distance_matrix,
1561     number_of_stops=number_of_stops,
1562     iterations=number_of_iterations,
1563     seed=seed,
1564     neighbourhood_operator=swap_stops,
1565 )
1566 print(solution_with_swap_stops)

```

giving:

Python output

```

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

```

Python input

```

1568 solution_with_reverse_path = run_neighbourhood_search(
1569     distance_matrix=distance_matrix,
1570     number_of_stops=number_of_stops,
1571     iterations=number_of_iterations,
1572     seed=seed,
1573     neighbourhood_operator=reverse_path,
1574 )
1575 print(solution_with_reverse_path)

```

giving:

Python output

```

1576 [0, 8, 5, 3, 1, 9, 12, 11, 4, 10, 6, 2, 7, 0]

```

Importantly, the costs differ substantially:

Python input

```

1577 cost = get_cost(
1578     tour=solution_with_swap_stops,
1579     distance_matrix=distance_matrix,
1580 )
1581 print(cost)

```

which gives:

Python output

```

1582 362

```

Whereas using the the reverse path operator, which corresponds to an algorithm called the “2 opt” algorithm, gives a lower cost:

Python input

```

1583 cost = get_cost(
1584     tour=solution_with_reverse_path,
1585     distance_matrix=distance_matrix,
1586 )
1587 print(cost)

```

which gives:

Python output

```

1588 299

```

9.4 SOLVING WITH R

To solve this problem using R we will write functionality that matches 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:

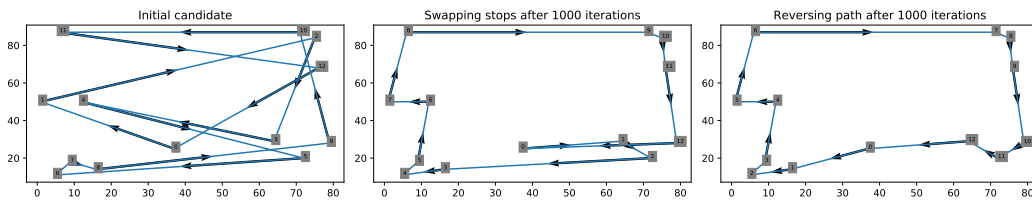


Figure 9.4 The final tours obtained by using the neighbourhood search in Python.

R input

```

1589 #' Return an initial tour.
1590 #'
1591 #' @param number_of_stops The number of stops.
1592 #' @param seed An integer seed. If an integer value is
1593 #'           passed it will create a random tour.
1594 #'
1595 #' @return A tour starting and ending at stop with index 0.
1596 get_initial_candidate <- function(number_of_stops, seed = NA){
1597   internal_stops <- 1:(number_of_stops - 1)
1598   if (!is.na(seed)) {
1599     set.seed(seed)
1600     internal_stops <- sample(internal_stops)
1601   }
1602   c(0, internal_stops, 0)
1603 }

```

Using this we can get a random tour on 13 stops:

R input

```

1604 number_of_stops <- 13
1605 seed <- 0
1606 initial_candidate <- get_initial_candidate(
1607   number_of_stops = number_of_stops,
1608   seed = seed)
1609 print(initial_candidate)

```

R output

1610

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

To be able to evaluate any given tour we see that we must also be able to evaluate its cost. Here we define `get_cost` to do this:

R input

1611

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

1612

```
##
```

1613

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

1614

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

1615

```
##
```

1616

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

1617

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

1618

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

1619

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

1620

```
}
```


R input

```

1621 distance_matrix <- rbind(
1622     c(0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1623     c(35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1624     c(35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1625     c(29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1626     c(70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1627     c(35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1628     c(42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1629     c(27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1630     c(24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1631     c(44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1632     c(58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1633     c(71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1634     c(69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0)
1635 )
1636 cost <- get_cost(
1637     tour = initial_candidate,
1638     distance_matrix = distance_matrix)
1639 print(cost)

```

R output

```

1640 [1] 709

```

We will now define two different neighbourhood operators:

- `swap_stops`: this swaps two stops in a given tour.
- `reverse_path`: this swaps two stops and reverts the stops in between them.

R input

```

1641 #' Return a new tour by swapping two stops.
1642 #'
1643 #' @param tour A given vector of successive stops.
1644 #'
1645 #' @return A tour
1646 swap_stops <- function(tour){
1647   number_of_stops <- length(tour) - 1
1648   stops_to_swap <- sort(sample(2:number_of_stops, 2))
1649   new_tour <- replace(x = tour,
1650                      list = stops_to_swap,
1651                      values = rev(tour[stops_to_swap]))
1652 }
1653
1654 #' Return a new tour by reversing the path between two stops.
1655 #'
1656 #' @param tour A given vector of successive stops.
1657 #'
1658 #' @return A tour
1659 reverse_path <- function(tour){
1660   number_of_stops <- length(tour) - 1
1661   stops_to_swap <- sort(sample(2:number_of_stops, 2))
1662   i <- stops_to_swap[1]
1663   j <- stops_to_swap[2]
1664   new_order <- c(
1665     c(1: (i - 1)),
1666     c(j:i),
1667     c( (j + 1): length(tour))
1668   )
1669   tour[new_order]
1670 }

```

If we apply these two neighbour operators to our initial candidate we can see the effects:

R input

```

1671 print(swap_stops(initial_candidate))

```

which swaps the 6th and 11th stops:

R output

```
1672 [1] 0 9 4 7 1 11 5 3 8 6 2 12 10 0
```

R input

```
1673 print(reverse_path(initial_candidate))
```

which reverses the order between the 7th and the 11th stop:

R output

```
1674 [1] 0 9 4 7 1 2 11 6 8 3 5 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

```

1675 #' Returns a tour by carrying out a neighbourhood search
1676 #'
1677 #' @param distance_matrix: the distance matrix
1678 #' @param number_of_stops: the number of stops
1679 #' @param iterations: the number of iterations for
1680 #'                      which to run the algorithm
1681 #' @param seed: a random seed (default: None)
1682 #' @param neighbourhood_operator: the neighbourhood operation
1683 #'                                (default: reverse_path)
1684 #'
1685 #' @return A tour
1686 run_neighbourhood_search <- function(
1687   distance_matrix,
1688   number_of_stops,
1689   iterations,
1690   seed = NA,
1691   neighbourhood_operator = reverse_path
1692 ){
1693   candidate <- get_initial_candidate(
1694     number_of_stops = number_of_stops,
1695     seed = seed
1696   )
1697
1698   best_cost <- get_cost(
1699     tour = candidate,
1700     distance_matrix = distance_matrix
1701   )
1702
1703   for (repetition in 1:iterations) {
1704     new_candidate <- neighbourhood_operator(candidate)
1705     cost <- get_cost(
1706       tour = new_candidate,
1707       distance_matrix = distance_matrix)
1708
1709     if (cost <= best_cost) {
1710       best_cost <- cost
1711       candidate <- new_candidate
1712     }
1713
1714   }
1715   candidate
1716 }

```

Using this we can see the effect of running 1000 iterations using different neighbourhood functions:

R input

```

1717 number_of_iterations <- 1000
1718 solution_with_swap_stops <- run_neighbourhood_search(
1719     distance_matrix = distance_matrix,
1720     number_of_stops = number_of_stops,
1721     iterations = number_of_iterations,
1722     seed = seed,
1723     neighbourhood_operator = swap_stops
1724 )
1725 print(solution_with_swap_stops)

```

giving:

R output

```

1726 [1] 0 11 4 10 6 2 7 8 5 3 1 9 12 0

```

R input

```

1727 number_of_iterations <- 1000
1728 solution_with_reverse_path <- run_neighbourhood_search(
1729     distance_matrix = distance_matrix,
1730     number_of_stops = number_of_stops,
1731     iterations = number_of_iterations,
1732     seed = seed,
1733     neighbourhood_operator = reverse_path
1734 )
1735 print(solution_with_reverse_path)

```

giving:

R output

```

1736 [1] 0 8 5 3 1 9 12 11 4 10 6 2 7 0

```

Importantly, the costs differ substantially:

R input

```
1737 cost <- get_cost(  
1738     tour = solution_with_swap_stops,  
1739     distance_matrix = distance_matrix  
1740 )  
1741 print(cost)
```

which gives:

R output

```
1742 [1] 373
```

Whereas using the reverse path operator, which corresponds to an algorithm called the “2 opt” algorithm, gives a lower cost:

R input

```
1743 cost <- get_cost(  
1744     tour = solution_with_reverse_path,  
1745     distance_matrix = distance_matrix  
1746 )  
1747 print(cost)
```

which gives:

R output

```
1748 [1] 299
```

9.5 RESEARCH

TBA

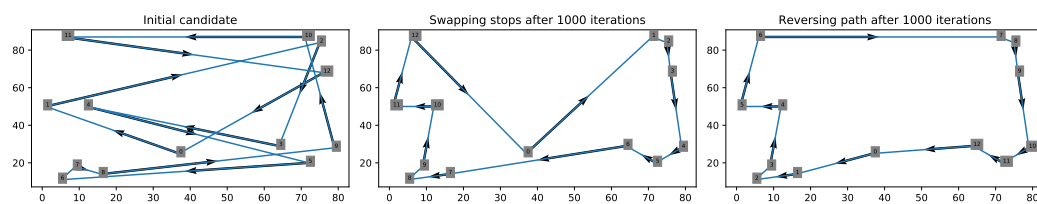


Figure 9.5 The final tours obtained by using the neighbourhood search in R



Bibliography

