

---

*Half Title Page*



---

# *Title Page*

*LOC Page*

*Vince: to Riggins*  
*Geraint: also, to Riggins*



---

# Contents

---

Contributors	xi
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?	4
1.3 WHAT IS OPEN SOURCE SOFTWARE	4
1.4 HOW TO GET THE MOST OUT OF THIS BOOK	5
1.5 HOW CODE IS WRITTEN IN THIS BOOK	6
SECTION II Probabilistic Modelling	
CHAPTER 2 ■ Markov Chains	11
2.1 PROBLEM	11
2.2 THEORY	11
2.3 SOLVING WITH PYTHON	14
2.4 SOLVING WITH R	19
2.5 WIDER CONTEXT	26
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 WIDER CONTEXT	43

## SECTION III Dynamical Systems

**CHAPTER 4 ■ Differential Equations** 47

---

4.1	PROBLEM	47
4.2	THEORY	47
4.3	SOLVING WITH PYTHON	48
4.4	SOLVING WITH R	53
4.5	WIDER CONTEXT	56

**CHAPTER 5 ■ Systems dynamics** 59

---

5.1	PROBLEM	59
5.2	THEORY	59
5.3	SOLVING WITH PYTHON	62
5.4	SOLVING WITH R	69
5.5	WIDER CONTEXT	75

## SECTION IV Emergent Behaviour

**CHAPTER 6 ■ Game Theory** 79

---

6.1	PROBLEM	79
6.2	THEORY	79
6.3	SOLVING WITH PYTHON	82
6.4	SOLVING WITH R	86
6.5	WIDER CONTEXT	89

**CHAPTER 7 ■ Agent Based Simulation** 91

---

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

## SECTION V Optimisation

**CHAPTER 8 ■ Linear programming** 109

---



8.1	PROBLEM	109
8.2	THEORY	110
8.3	SOLVING WITH PYTHON	114
8.4	SOLVING WITH R	119
8.5	WIDER CONTEXT	128
CHAPTER	9 ■ Heuristics	129
9.1	PROBLEM	129
9.2	THEORY	129
9.3	SOLVING WITH PYTHON	131
9.4	SOLVING WITH R	139
9.5	WIDER CONTEXT	146
	Bibliography	149



---

# Contributors

---

**Geraint Palmer**

Cardiff University School of Mathematics  
Cardiff, Wales, UK

**Vincent Knight**

Cardiff University School of Mathematics  
Cardiff, Wales, UK



# 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?

---

This book is aimed at readers who want to use open source software to solve the considered applied mathematical problems.

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 to be able to download the relevant software;
- Have done any introductory tutorial in the languages they plan to use;
- 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.

By reading a particular chapter of the book, the reader will have:

## 4 ■ Applied Mathematics Problems with Open Source Software

1. the practical knowledge to solve problems using a computer;
2. an overview of the higher level theoretic concepts;
3. pointers to further reading to gain background understand and research undertaken using the concepts.

### 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 Pokémon. Using an area of mathematics call linear programming which is based on linear algebra he was able to find the best makeup of Pokémon.

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 over the world and powers some of the most important infrastructure around. A good example of this is cryptographic software which should not rely on secrecy for security<sup>1</sup>. This implies that cryptographic systems that do not require trust in a hidden system can exist. In practice these are all 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 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

---

<sup>1</sup>This is also referred to as Kerckhoffs's principle which states that "a cryptosystem should be secure, even if everything about the system, except the key, is public knowledge" (Auguste Kerckhoffs. "La cryptographie militaire. *Journal des sciences militaires*". In: *IX (38)* 5 [1883])



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 shoulder 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](https://github.com/drvinceknight/ampwoss). There 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 a particular problem and wanting to know where to start.

After this introductory chapter the book is split in to 4 sections. Each section corresponds to a broad problem type and contains 2 chapters that correspond to 2 solution approaches. The first chapter in a section is based on exact methodology whereas the second chapter is based on heuristic methodology. The structure of the book is:

1. Probabilistic modelling
  - Markov chains
  - Discrete event simulation
2. Dynamical systems
  - Differential equations
  - Systems dynamics
3. Emergent behaviour
  - Game theory.
  - Agent based simulation
4. Optimisation.
  - Linear programming
  - Heuristics

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. An overview of the theory as well as a discussion as to how the theory relates to the considered problem. Readers will also be presented with reference texts if they want to gain a more in depth understanding.
4. Solving with Python. We will describe how to use tools available in Python to solve the problem.
5. Solving with R. We will describe how to use tools available in R to solve the problem.
6. This section will include a few hand picked academic papers relevant to the covered topic. It is hoped that these few papers can be a good starting point for someone wanting to not only use the methodology described but also understand the broader field.

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. The R and Python sections are **purposefully** written as near clones of each other so that a reader can read only the section that interests them. In places there are some minor differences in the text and this is due to differences of implementation in the respective languages.

Please do take from the book what you find useful.

## 1.5 HOW CODE IS WRITTEN IN THIS BOOK

---

Throughout this book, there are going to be various pieces of code written. Code is a series of instructions that usually give some sort of output when submitted to a computer.

This book will show both the set of instructions (referred to as the input) and the output.

You will see Python input as follows:

### Python input

```
1 print(2 + 2)
```

and you will see Python output as follows:

### Python output

```
2 4
```

You will see R input as follows:

### R input

```
3 print(2 + 2)
```

and you will see R output as follows:

### R output

```
4 [1] 4
```

As well as this, a continuous line numbering across all code sections is used so that if the reader needs to refer to a given set of input or output this can be done.

The code itself is written using 3 principles:

- Modularity: code is written as a series of smaller sections of code. These correspond to smaller, simpler, individual tasks (modules) that can be used together to carry out a particular larger task.
- Documentation: readable variable names as well as text describing the functionality of each module of code are used throughout. This ensures that code is not only usable but also understandable.
- Tests: there are places where each module of code is used independently to check the output. This can be thought of as a test of functionality which readers can use to check they are getting expected results.

These are best practice principles in research software development that ensure usable, reproducible and reliable code.<sup>2</sup> Interested readers might want to see Figure 1.1 which shows how the 3 principles interact with each other.

---

<sup>2</sup>Greg Wilson et al. “Best practices for scientific computing”. In: *PLoS biology* 12.1 (2014), e1001745.

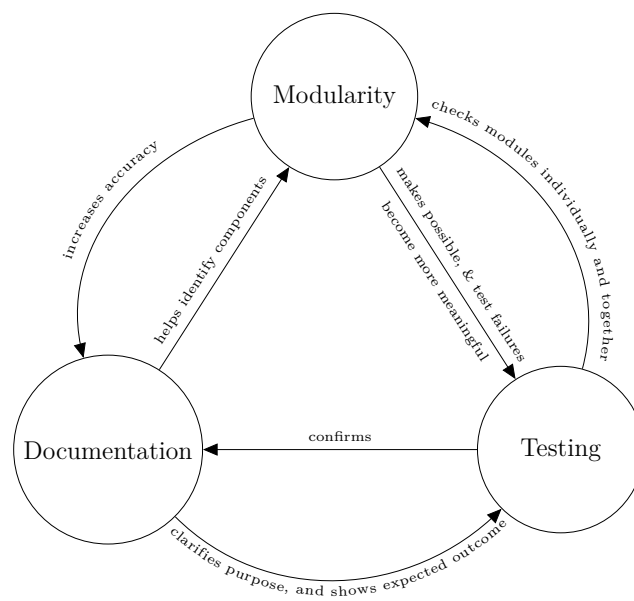


Figure 1.1 The relationship between modularisation, documentation and testing. The authors thank Dr Nikoleta Glynatsi for their contribution to the drawing of this diagram.

# II

---

## Probabilistic Modelling



# Markov Chains

---

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

## 2.1 PROBLEM

---

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

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

This is represented diagrammatically in Figure 2.1.

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

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

## 2.2 THEORY

---

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

For example, in the barber shop a single number is sufficient to describe the status of the shop: the number of customers present. If that number is 1 this implies that

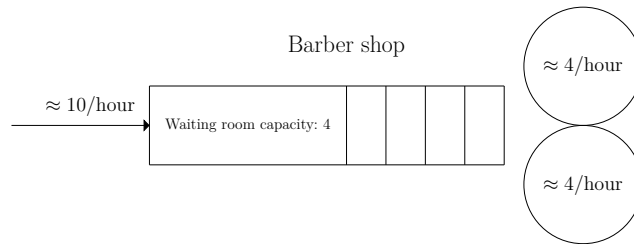


Figure 2.1 Diagrammatic representation of the barber shop as a queuing system.

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

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

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

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

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

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

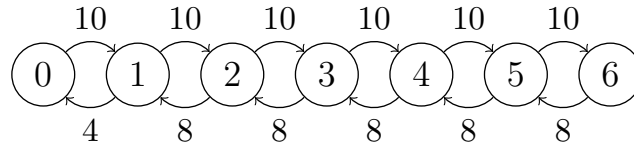


Figure 2.2 Diagrammatic representation of the state space and the transition rates

Note that a Markov chain assumes the rates follow an exponential distribution. One interesting property of this distribution is that it is considered memoryless which means the probability of a customer finishing service within the next 5 minutes does not change if they have been having their hair cut for 3 minutes already.



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

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

You will see that  $Q_{ii}$  are negative and ensure the rows of  $Q$  sum to 0. This gives the total rate of change leaving state  $i$ .

The matrix  $Q$  can be used to understand the probability of being in a given state after  $t$  time units. This can be represented mathematically using a matrix  $P_t$  where  $(P_t)_{ij}$  is the probability of being in state  $j$  after  $t$  time units having started in state  $i$ . Using a mathematical tool called the matrix exponential<sup>1</sup>

the value of  $P_t$  can be calculated numerically.

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

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

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

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

with the following constraint:

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

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

---

1

Chapter 9 of the following text book give a description of how to compute the matrix exponential numerically (Charles F Van Loan and G Golub. *Matrix computations (Johns Hopkins studies in mathematical sciences)*. The Johns Hopkins University Press, 1996) and (Cleve Moler and Charles Van Loan. “Nineteen dubious ways to compute the exponential of a matrix”. In: *SIAM review* 20.4 [1978], pp. 801–836; Cleve Moler and Charles Van Loan. “Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later”. In: *SIAM review* 45.1 [2003], pp. 3–49) give a review of 19 algorithms that can be used.

## 2.3 SOLVING WITH PYTHON

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

### Python input

```

5  def get_transition_rate(
6      in_state,
7      out_state,
8      waiting_room=4,
9      num_barbers=2,
10 ):
11     """Return the transition rate for 2 given states.
12
13     Args:
14         in_state: an integer
15         out_state: an integer
16         waiting_room: an integer (default: 4)
17         num_barbers: an integer (default: 2)
18
19     Returns:
20         A real.
21     """
22     arrival_rate = 10
23     service_rate = 4
24
25     capacity = waiting_room + num_barbers
26     delta = out_state - in_state
27
28     if delta == 1 and in_state < capacity:
29         return arrival_rate
30
31     if delta == -1:
32         return min(in_state, num_barbers) * service_rate
33
34     return 0

```

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

## Python input

```

35 import itertools
36 import numpy as np
37
38
39 def get_transition_rate_matrix(waiting_room=4, num_barbers=2):
40     """Return the transition matrix Q.
41
42     Args:
43         waiting_room: an integer (default: 4)
44         num_barbers: an integer (default: 2)
45
46     Returns:
47         A matrix.
48     """
49     capacity = waiting_room + num_barbers
50     state_pairs = itertools.product(
51         range(capacity + 1), repeat=2
52     )
53
54     flat_transition_rates = [
55         get_transition_rate(
56             in_state=in_state,
57             out_state=out_state,
58             waiting_room=waiting_room,
59             num_barbers=num_barbers,
60         )
61         for in_state, out_state in state_pairs
62     ]
63     transition_rates = np.reshape(
64         flat_transition_rates, (capacity + 1, capacity + 1)
65     )
66     np.fill_diagonal(
67         transition_rates, -transition_rates.sum(axis=1)
68     )
69
70     return transition_rates

```

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

**Python input**

```

71 Q = get_transition_rate_matrix()
72 print(Q)

```

which gives:

**Python output**

```

73 [[-10  10   0   0   0   0   0]
74 [  4 -14  10   0   0   0   0]
75 [  0   8 -18  10   0   0   0]
76 [  0   0   8 -18  10   0   0]
77 [  0   0   0   8 -18  10   0]
78 [  0   0   0   0   8 -18  10]
79 [  0   0   0   0   0   8 -8]]

```

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

**Python input**

```

80 import scipy.linalg
81
82 print(scipy.linalg.expm(Q * 0.5).round(5))

```

which gives:

**Python output**

```

83 [[0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.0815 ]
84 [0.08501 0.18292 0.18666 0.1708  0.14377 0.1189  0.11194]
85 [0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346]
86 [0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142]
87 [0.02667 0.07361 0.10005 0.13422 0.17393 0.2189  0.27262]
88 [0.01567 0.0487  0.07552 0.11775 0.17512 0.24484 0.32239]
89 [0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914]]

```

To see what would happen after 500 time units:

**Python input**

```

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

```

which gives:

### Python output

```

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]
94  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
95  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
96  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
97  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]]

```

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

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

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

### Python input

```

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

```

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

### Python input

```

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

```

giving:

### Python output

```
114 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
```

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

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

### Python input

```
115 def get_probability_of_full_shop(
116     waiting_room=4, num_barbers=2
117 ):
118     """Return the probability of the barber shop being full.
119
120     Args:
121         waiting_room: an integer (default: 4)
122         num_barbers: an integer (default: 2)
123
124     Returns:
125         A real.
126     """
127     Q = get_transition_rate_matrix(
128         waiting_room=waiting_room,
129         num_barbers=num_barbers,
130     )
131     pi = get_steady_state_vector(Q)
132     return pi[-1]
```

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

### Python input

```
133 print(round(get_probability_of_full_shop(), 6))
```

which gives:

### Python output

```
134 0.261756
```

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

Having 2 extra space in the waiting room corresponds to:

#### Python input

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

which gives:

#### Python output

```
136 0.23557
```

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

#### Python input

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

#### Python output

```
138 0.078636
```

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

## 2.4 SOLVING WITH R

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

**R input**

```

139 #' Return the transition rate for 2 given states.
140 #'
141 #' @param in_state an integer
142 #' @param out_state an integer
143 #' @param waiting_room an integer (default: 4)
144 #' @param num_barbers an integer (default: 2)
145 #'
146 #' @return A real
147 get_transition_rate <- function(in_state,
148                                out_state,
149                                waiting_room = 4,
150                                num_barbers = 2){
151   arrival_rate <- 10
152   service_rate <- 4
153
154   capacity <- waiting_room + num_barbers
155   delta <- out_state - in_state
156
157   if (delta == 1) {
158     if (in_state < capacity) {
159       return(arrival_rate)
160     }
161   }
162
163   if (delta == -1) {
164     return(min(in_state, num_barbers) * service_rate)
165   }
166   return(0)
167 }

```

This actual function will not be used but instead a vectorized version<sup>2</sup> of this makes calculations more efficient:

<sup>2</sup>A vectorized calculation refers to the manner in which an instruction is given to a computer. When vectorized: a single instruction with multiple data are given at the same time which corresponds to "Single instruction, multiple data" (SIMD) as defined in Flynn's taxonomy (Michael J Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 [1966], pp. 1901–1909). This is a type of parallelisation that can be done at the central processing unit level of the computer.



**R input**

```

168 vectorized_get_transition_rate <- Vectorize(
169   get_transition_rate,
170   vectorize.args = c("in_state", "out_state")
171 )

```

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**

```

172 #' Return the transition rate matrix Q
173 #'
174 #' @param waiting_room an integer (default: 4)
175 #' @param num_barbers an integer (default: 2)
176 #'
177 #' @return A matrix
178 get_transition_rate_matrix <- function(waiting_room = 4,
179                                       num_barbers = 2){
180   max_state <- waiting_room + num_barbers
181
182   Q <- outer(0:max_state,
183             0:max_state,
184             vectorized_get_transition_rate,
185             waiting_room = waiting_room,
186             num_barbers = num_barbers
187           )
188   row_sums <- rowSums(Q)
189
190   diag(Q) <- -row_sums
191   Q
192 }

```

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

**R input**

```

193 Q <- get_transition_rate_matrix()
194 print(Q)

```

which gives:

**R output**

```

195      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
196 [1,]  -10  10   0   0   0   0   0
197 [2,]   4 -14  10   0   0   0   0
198 [3,]   0   8 -18  10   0   0   0
199 [4,]   0   0   8 -18  10   0   0
200 [5,]   0   0   0   8 -18  10   0
201 [6,]   0   0   0   0   8 -18  10
202 [7,]   0   0   0   0   0   8  -8

```

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

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

**R input**

```

203 library(expm, warn.conflicts = FALSE, quietly = TRUE)
204 library(magrittr, warn.conflicts = FALSE, quietly = TRUE)
205
206 print( (Q * .5) %>% expm %>% round(5))

```

which gives:

**R output**

```

207      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
208 [1,] 0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.08150
209 [2,] 0.08501 0.18292 0.18666 0.17080 0.14377 0.11890 0.11194
210 [3,] 0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346
211 [4,] 0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142
212 [5,] 0.02667 0.07361 0.10005 0.13422 0.17393 0.21890 0.27262
213 [6,] 0.01567 0.04870 0.07552 0.11775 0.17512 0.24484 0.32239
214 [7,] 0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914

```

After 500 time units:

**R input**

```

215 print( (Q * 500) %>% expm %>% round(5))

```

which gives:

## R output

```

216      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
217 [1,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
218 [2,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
219 [3,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
220 [4,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
221 [5,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
222 [6,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
223 [7,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176

```

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

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

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

## R input

```

224 library(pracma, warn.conflicts = FALSE, quietly = TRUE)
225
226 #' Return the steady state vector of any given continuous time
227 #' transition rate matrix
228 #'
229 #' @param Q a transition rate matrix
230 #'
231 #' @return A vector
232 get_steady_state_vector <- function(Q){
233   state_space_size <- dim(Q)[1]
234   A <- rbind(t(Q), 1)
235   b <- c(integer(state_space_size), 1)
236   mldivide(A, b)
237 }

```

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

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

**R input**

```
238 print(get_steady_state_vector(Q))
```

giving:

**R output**

```
239           [,1]
240 [1,] 0.03430888
241 [2,] 0.08577220
242 [3,] 0.10721525
243 [4,] 0.13401906
244 [5,] 0.16752383
245 [6,] 0.20940479
246 [7,] 0.26175598
```

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

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

**R input**

```
247 #' Return the probability of the barber shop being full
248 #'
249 #' @param waiting_room (default: 4)
250 #' @param num_barbers (default: 2)
251 #'
252 #' @return A real
253 get_probability_of_full_shop <- function(waiting_room = 4,
254                                         num_barbers = 2){
255   arrival_rate <- 10
256   service_rate <- 4
257   pi <- get_transition_rate_matrix(
258     waiting_room = waiting_room,
259     num_barbers = num_barbers
260   ) %>%
261     get_steady_state_vector()
262
263   capacity <- waiting_room + num_barbers
264   pi[capacity + 1]
265 }
```

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

#### R input

```
266 print(get_probability_of_full_shop())
```

which gives:

#### R output

```
267 [1] 0.261756
```

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

Adding 2 extra spaces in the waiting rooms corresponds to:

#### R input

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

which decreases the probability of a full shop to:

#### R output

```
269 [1] 0.2355699
```

but adding another barber and chair:

#### R input

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

gives:

#### R output

```
271 [1] 0.0786359
```

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

## 2.5 WIDER CONTEXT

---

The overview of Markov chains given here has mainly concentrated on calculation of steady state probabilities. There are in fact many more theoretic and applied aspects of Markov chain models. Some examples of this include the calculation of sojourn times which is how long a system spends in a given state as well as considering models with absorption: where the system arrives at a state that it no longer leaves. For a good overview of these the following textbook is recommended: (William J Stewart. *Probability, Markov chains, queues, and simulation*. Princeton university press, 2009).

In (Bari Tan. “Markov chains and the RISK board game”. In: *Mathematics Magazine* 70.5 [1997], pp. 349–357; Ian Stewart. “Monopoly revisited”. In: *Scientific American* 275.4 [1996], pp. 116–119), Markov chains are used to model board games. In (Tan, “Markov chains and the RISK board game”) a model of the battles that take place on a Risk board is used to understand the probabilities of invasion of territories based on troupe numbers. This is done using an absorbing Markov chain. In (Stewart, “Monopoly revisited”) a standard model is used to identify the properties that are most likely to be landed on in Monopoly. This is done through calculation of steady state probabilities. These are both examples of discrete time Markov chains.

A common application of Markov chains is in queueing systems and specifically queueing systems applied to healthcare. In (Jeff D Griffiths, Janet E Williams, and Richard Max Wood. “Modelling activities at a neurological rehabilitation unit”. In: *European Journal of Operational Research* 226.2 [2013], pp. 301–312) a model of a neurological rehabilitation unit is built and used to help better staff the unit. This is accomplished using the steady state probabilities and calculating various performance measures. This is an application of a continuous time Markov chain.

An extension of Markov chains are Markov decision processes. This is a particular mathematical model that identifies the optimal decision made within a Markov chain. Instead of building multiple Markov models for different decisions, in Markov decision processes decisions can be made at each state of the underlying chain. A policy can be identified giving the optimal decision at each state. In (Douglas J White. “A survey of applications of Markov decision processes”. In: *Journal of the operational research society* 44.11 [1993], pp. 1073–1096) a literature review is given showing a wide ranging application of these decision processes from to agriculture to motor insurance claims as well as sports.

# 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?

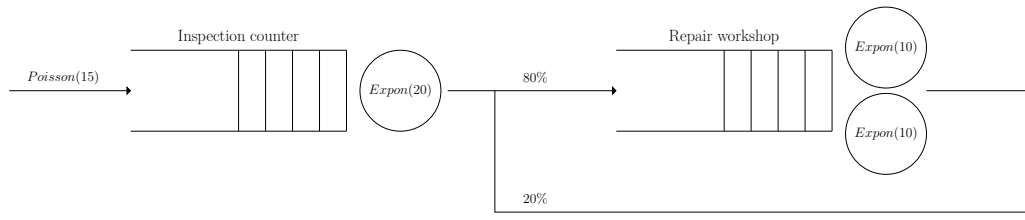


Figure 3.1 Diagrammatic representation of the bicycle repair shop as a queuing system.

### 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<sup>1</sup>. 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.

---

<sup>1</sup>An early discussion of pseudo random numbers is (John Von Neumann. “13. various techniques used in connection with random digits”. In: *Appl. Math Ser* 12.36-38 [1951], p. 3) where the author claimed: “Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” A number of different pseudo random number generators exist, at the time of writing the state of the art is the Mersenne Twister described in (Makoto Matsumoto and Takuji Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 [1998], pp. 3–30).

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.<sup>2</sup>

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:

---

<sup>2</sup>William J Stewart. *Probability, Markov chains, queues, and simulation*. Princeton university press, 2009.

## Python input

```

272 import ciw
273
274
275 def build_network_object(
276     num_inspectors=1,
277     num_repairers=2,
278 ):
279     """Returns a Network object that defines the repair shop.
280
281     Args:
282         num_inspectors: a positive integer (default: 1)
283         num_repairers: a positive integer (default: 2)
284
285     Returns:
286         a Ciw network object
287     """
288     arrival_rate = 15
289     inspection_rate = 20
290     repair_rate = 10
291     prob_need_repair = 0.8
292     N = ciw.create_network(
293         arrival_distributions=[
294             ciw.dists.Exponential(arrival_rate),
295             ciw.dists.NoArrivals(),
296         ],
297         service_distributions=[
298             ciw.dists.Exponential(inspection_rate),
299             ciw.dists.Exponential(repair_rate),
300         ],
301         number_of_servers=[num_inspectors, num_repairers],
302         routing=[[0.0, prob_need_repair], [0.0, 0.0]],
303     )
304     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

```

305 N = build_network_object()
306 print(N.number_of_nodes)

```

which gives:

### Python output

```
307 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

```
308 def run_simulation(network, seed=0):
309     """Builds a simulation object and runs it for 8 time units.
310
311     Args:
312         network: a Ciw network object
313         seed: a float (default: 0)
314
315     Returns:
316         a Ciw simulation object after a run of the simulation
317     """
318     max_time = 8
319     ciw.seed(seed)
320     Q = ciw.Simulation(network)
321     Q.simulate_until_max_time(max_time)
322     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<sup>3</sup>. 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:

<sup>3</sup>Pseudo random number generators give a sequence of numbers that obey a series of properties. A seed is necessary to obtain a starting point for a given sequence. This has the benefit of ensuring that given sequences can be reproduced.

## Python input

```

323 import pandas as pd
324
325
326 def get_proportion(Q):
327     """Returns the proportion of bicycles spending over a given
328     limit at the repair shop.
329
330     Args:
331         Q: a Ciw simulation object after a run of the
332             simulation
333
334     Returns:
335         a real
336     """
337     limit = 0.5
338     inds = Q.nodes[-1].all_individuals
339     recs = pd.DataFrame(
340         dr for ind in inds for dr in ind.data_records
341     )
342     recs["total_time"] = (
343         recs["exit_date"] - recs["arrival_date"]
344     )
345     total_times = recs.groupby("id_number")["total_time"].sum()
346     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

```

347 N = build_network_object()
348 Q = run_simulation(N)
349 p = get_proportion(Q)
350 print(round(p, 6))

```

This gives:

## Python output

```

351 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

```

352 def get_average_proportion(num_inspectors=1, num_repairers=2):
353     """Returns the average proportion of bicycles spending over
354     a given limit at the repair shop.
355
356     Args:
357         num_inspectors: a positive integer (default: 1)
358         num_repairers: a positive integer (default: 2)
359
360     Returns:
361         a real
362     """
363     num_trials = 100
364     N = build_network_object(
365         num_inspectors=num_inspectors,
366         num_repairers=num_repairers,
367     )
368     proportions = []
369     for trial in range(num_trials):
370         Q = run_simulation(N, seed=trial)
371         proportion = get_proportion(Q=Q)
372         proportions.append(proportion)
373     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

```

374 p = get_average_proportion(num_inspectors=1, num_repairers=2)
375 print(round(p, 6))

```

which gives:

### Python output

```

376 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

```
377 p = get_average_proportion(num_inspectors=2, num_repairers=2)
378 print(round(p, 6))
```

which gives:

#### Python output

```
379 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

```
380 p = get_average_proportion(num_inspectors=1, num_repairers=3)
381 print(round(p, 6))
```

which gives:

#### Python output

```
382 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



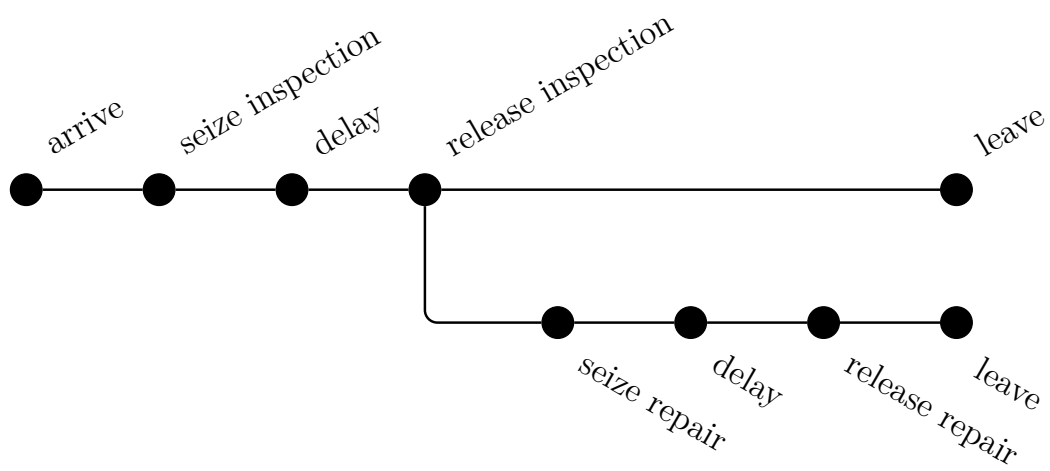


Figure 3.2 Diagrammatic representation of the forked trajectories a bicycle can take

in Figure 3.2 shows the branched trajectories than a bicycle would take at the repair shop:

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

## R input

```

383 library(simmer)
384
385 #' Returns a simmer trajectory object outlining the bicycles
386 #' path through the repair shop
387 #'
388 #' @return A simmer trajectory object
389 define_bicycle_trajectories <- function() {
390   inspection_rate <- 20
391   repair_rate <- 10
392   prob_need_repair <- 0.8
393   bicycle <-
394     trajectory("Inspection") %>%
395     seize("Inspector") %>%
396     timeout(function() {
397       rexp(1, inspection_rate)
398     }) %>%
399     release("Inspector") %>%
400     branch(
401       function() (runif(1) < prob_need_repair),
402       continue = c(F),
403       trajectory("Repair") %>%
404         seize("Repairer") %>%
405         timeout(function() {
406           rexp(1, repair_rate)
407         }) %>%
408         release("Repairer"),
409       trajectory("Out")
410     )
411   return(bicycle)
412 }

```

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

```

413 #' Runs one trial of the simulation.
414 #'
415 #' @param bicycle a simmer trajectory object
416 #' @param num_inspectors positive integer (default: 1)
417 #' @param num_repairers positive integer (default: 2)
418 #' @param seed a float (default: 0)
419 #'
420 #' @return A simmer simulation object after one run of
421 #'         the simulation
422 run_simulation <- function(bicycle,
423                           num_inspectors = 1,
424                           num_repairers = 2,
425                           seed = 0) {
426   arrival_rate <- 15
427   max_time <- 8
428   repair_shop <-
429     simmer("Repair Shop") %>%
430     add_resource("Inspector", num_inspectors) %>%
431     add_resource("Repairer", num_repairers) %>%
432     add_generator("Bicycle", bicycle, function() {
433       rexp(1, arrival_rate)
434     })
435
436   set.seed(seed)
437   repair_shop %>% run(until = 8)
438   return(repair_shop)
439 }

```

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<sup>4</sup>. 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:

<sup>4</sup>Pseudo random number generators give a sequence of numbers that obey a series of properties. A seed is necessary to obtain a starting point for a given sequence. This has the benefit of ensuring that given sequences can be reproduced.

**R input**

```

440 #' Returns the proportion of bicycles spending over 30
441 #' minutes in the repair shop
442 #'
443 #' @param repair_shop a simmer simulation object
444 #'
445 #' @return a float between 0 and 1
446 get_proportion <- function(repair_shop) {
447   limit <- 0.5
448   recs <- repair_shop %>% get_mon_arrivals()
449   total_times <- recs$end_time - recs$start_time
450   return(mean(total_times > 0.5))
451 }

```

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**

```

452 bicycle <- define_bicycle_trajectories()
453 repair_shop <- run_simulation(bicycle = bicycle)
454 print(get_proportion(repair_shop = repair_shop))

```

This piece of code gives

**R output**

```

455 [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**

```

456 #' Returns the average proportion of bicycles spending over
457 #' a given limit at the repair shop.
458 #'
459 #' @param num_inspectors positive integer (default: 1)
460 #' @param num_repairers positive integer (default: 2)
461
462 #' @return a float between 0 and 1
463 get_average_proportion <- function(num_inspectors = 1,
464                                   num_repairers = 2) {
465   num_trials <- 100
466   bicycle <- define_bicycle_trajectories()
467   proportions <- c()
468   for (trial in 1:num_trials) {
469     repair_shop <- run_simulation(
470       bicycle = bicycle,
471       num_inspectors = num_inspectors,
472       num_repairers = num_repairers,
473       seed = trial
474     )
475     proportion <- get_proportion(
476       repair_shop = repair_shop
477     )
478     proportions[trial] <- proportion
479   }
480   return(mean(proportions))
481 }

```

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

**R input**

```

482 print(
483   get_average_proportion(
484     num_inspectors = 1,
485     num_repairers = 2)
486 )

```

which gives:

**R output**

```
487 [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 situation where the additional member of staff works at the inspection desk:

**R input**

```
488 print(  
489   get_average_proportion(  
490     num_inspectors = 2,  
491     num_repairers = 2)  
492 )
```

which gives:

**R output**

```
493 [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**

```
494 print(  
495   get_average_proportion(  
496     num_inspectors = 1,  
497     num_repairers = 3)  
498 )
```

which gives:

**R output**

```
499 [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 WIDER CONTEXT

The concepts shown in this chapter cover some theoretical aspects of discrete event simulation. There are a number of further topics that can be vital to creating valid models of real life scenarios. These include time dependent rates and rostering servers. An overview of the theory of discrete event simulation is given in (Stewart Robinson. *Simulation: the practice of model development and use*. Vol. 50. Wiley Chichester, 2004).

One particular use of discrete event simulation is as part of a wider optimisation exercise. For example, a systematic search for an optimal service configuration can use a discrete event simulation model as a replacement for a mathematical objective function. Another approach is to integrate an optimisation procedure<sup>5</sup> within a discrete event simulation model so as to iteratively simulate optimal configurations. This is done in (Andres F Osorio et al. “Simulation-optimization model for production planning in the blood supply chain”. In: *Health care management science* 20.4 [2017], pp. 548–564) to be able to bring together strategic configuration and overall flow in the blood supply chain. A general review and taxonomy of different uses of discrete event simulation with optimisation techniques is given in (Gonçalo Figueira and Bernardo Almada-Lobo. “Hybrid simulation–optimization methods: A taxonomy and discussion”. In: *Simulation Modelling Practice and Theory* 46 [2014], pp. 118–134).

One domain where simulation is popularly used is in modelling healthcare systems. A general overview is given in (Sally C Brailsford et al. “An analysis of the academic literature on simulation and modelling in health care”. In: *Journal of simulation* 3.3 [2009], pp. 130–140) where uses include resource utilisation, human behaviour, and workforce management.

In order to be able to fully capture all the relevant details of the system to be modelled, an extension of discrete event simulation is to combine the methodology with systems dynamics (Chapter 5) in order to model continuous aspects of the system and/or agent based modelling (Chapter 7) in order to observe emergent or learned behaviours. This is known as hybrid simulation, and an overview is given in (Sally C. Brailsford et al. “Hybrid simulation modelling in operational research: A state-of-the-art review”. In: *European Journal of Operational Research* 278.3 [2019], pp. 721–737. ISSN: 0377-2217. DOI: <https://doi.org/10.1016/j.ejor.2018.10.025>. URL: <https://www.sciencedirect.com/science/article/pii/S0377221718308786>). There are a number of ways of combining these methodologies, from comparison to full integration.

<sup>5</sup>For more information on optimisation see Chapters 8 and 9.





# III

## Dynamical Systems



# Differential Equations

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

## 4.1 PROBLEM

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

## 4.2 THEORY

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

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

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

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

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

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

In this particular case, one such function is:

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

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

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

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

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

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

where  $K$  is the cost per person per unit time.

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

### 4.3 SOLVING WITH PYTHON

---

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

## Python input

```

500 import sympy as sym
501
502 t = sym.Symbol("t")
503 alpha = sym.Symbol("alpha")
504 I_0 = sym.Symbol("I_0")
505 I = sym.Function("I")
506
507
508 def get_equation(alpha=alpha):
509     """Return the differential equation.
510
511     Args:
512         alpha: a float (default: symbolic alpha)
513
514     Returns:
515         A symbolic equation
516     """
517     return sym.Eq(sym.Derivative(I(t), t), -alpha * I(t))

```

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

## Python input

```

518 eq = get_equation()
519 print(eq)

```

which gives:

## Python output

```

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

```

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

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

A value of  $\alpha$  can be passed if required:

**Python input**

```

521 eq = get_equation(alpha=1)
522 print(eq)

```

**Python output**

```

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

```

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

**Python input**

```

524 def get_solution(I_0=I_0, alpha=alpha):
525     """Return the solution to the differential equation.
526
527     Args:
528         I_0: a float (default: symbolic I_0)
529         alpha: a float (default: symbolic alpha)
530
531     Returns:
532         A symbolic equation
533     """
534     eq = get_equation(alpha=alpha)
535     return sym.dsolve(eq, I(t), ics={I(0): I_0})

```

This can verify the solution discussed previously:

**Python input**

```

536 sol = get_solution()
537 print(sol)

```

which gives:

**Python output**

```

538 Eq(I(t), I_0*exp(-alpha*t))

```

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

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

#### Python input

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

which gives:

#### Python output

```
540 True
```

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

#### Python input

```
541 sol = get_solution(alpha=2, I_0=100)
542 print(sol)
```

which gives:

#### Python output

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

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

## Python input

```

544 def get_cost(
545     I_0=I_0,
546     alpha=alpha,
547     cost_per_person=10,
548     cost_of_cure=0,
549 ):
550     """Return the cost.
551
552     Args:
553         I_0: a float (default: symbolic I_0)
554         alpha: a float (default: symbolic alpha)
555         cost_per_person: a float (default: 10)
556         cost_of_cure: a float (default: 0)
557
558     Returns:
559         A symbolic expression
560     """
561     I_sol = get_solution(I_0=I_0, alpha=alpha)
562     return (
563         sym.integrate(I_sol.rhs, (t, 0, sym.oo))
564         * cost_per_person
565         + cost_of_cure * I_0
566     )

```

The cost without purchasing the cure is:

## Python input

```

567 I_0 = 100
568 alpha = 2
569 cost_without_cure = get_cost(I_0=I_0, alpha=alpha)
570 print(cost_without_cure)

```

which gives:

## Python output

```

571 500

```

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



**Python input**

```

572 cost_of_cure = 5
573 cost_with_cure = get_cost(
574     I_0=I_0, alpha=2 * alpha, cost_of_cure=cost_of_cure
575 )
576 print(cost_with_cure)

```

which gives:

**Python output**

```

577 750

```

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

#### 4.4 SOLVING WITH R

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

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

**R input**

```

578 #' Returns the numerical value of the derivative.
579 #'
580 #' @param t a set of time points
581 #' @param y a function
582 #' @param parameters the set of all parameters passed to y
583
584 #' @return a float
585 derivative <- function(t, y, parameters) {
586     with(as.list(c(y, parameters)), {
587         dIdt <- -alpha * I # nolint
588         list(dIdt) # nolint
589     })
590 }

```

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

**R input**

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

This gives:

**R output**

```
592 [[1]]
593 [1] -200
```

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

**R input**

```
594 library(deSolve) # nolint
595 #' Return the solution to the differential equation.
596 #'
597 #' @param times: a vector of time points
598 #' @param y_0: a float (default: 100)
599 #' @param alpha: a float (default: 2)
600
601 #' @return A vector of numerical values
602 get_solution <- function(times,
603                           y0 = c(I = 100),
604                           alpha = 2) {
605   params <- c(alpha = alpha)
606   ode(y = y0, times = times, func = derivative, parms = params)
607 }
```

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

## R input

```

608 #' Return the cost.
609 #'
610 #' @param I_0: a float (default: symbolic I_0)
611 #' @param alpha: a float (default: symbolic alpha)
612 #' @param cost_per_person: a float (default: 10)
613 #' @param cost_of_cure: a float (default: 0)
614 #' @param step_size: a float (default: 0.0001)
615 #' @param max_time: an integer (default: 10)
616
617 #' @return A numeric value
618 get_cost <- function(
619     I_0 = 100,
620     alpha = 2,
621     cost_per_person = 10,
622     cost_of_cure = 0,
623     step_size = 0.0001,
624     max_time = 10) {
625   times <- seq(0, max_time, by = step_size)
626   out <- get_solution(times,
627     y0 = c(I = I_0),
628     alpha = alpha
629   )
630   number_of_observations <- length(out[, "I"])
631
632   time_between_steps <- diff(out[, "time"])
633   area_under_curve <- sum(
634     time_between_steps *
635     out[-number_of_observations, "I"]
636   )
637   area_under_curve *
638     cost_per_person + cost_of_cure *
639     I_0
640 }

```

The cost without purchasing the cure is:

## R input

```

641 alpha <- 2
642 cost_without_cure <- get_cost(alpha = alpha)
643 print(round(cost_without_cure))

```

which gives:

### R output

```
644 [1] 500
```

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

### R input

```
645 cost_of_cure <- 5
646 cost_with_cure <- get_cost(
647     alpha = 2 * alpha, cost_of_cure = cost_of_cure
648 )
649 print(round(cost_with_cure))
```

which gives:

### R output

```
650 [1] 750
```

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

## 4.5 WIDER CONTEXT

There are a number of further areas related to the study of as well as the use of differential equations. Topics omitted here include the actual solution approaches which in this chapter are taken care of using open source software. Chapters 9, 14 and 16 of (James Stewart. *Calculus: Concepts and contexts*. Cengage Learning, 2009) provide a good introduction to some of these concepts as well as a general discussion of the area of mathematics in which they sit: Calculus.

Differential equations have been applied in many settings. In (Frederick William Lanchester. *Aircraft in warfare: The dawn of the fourth arm*. Constable limited, 1916) differential equations were used to model attrition in warfare, the insights from these differential equations are referred to as Lanchester's square law. This has been historically fitted to a number of battles with varying levels of success.

In (Richard Syms and Laszlo Solymar. "A dynamic competition model of regime change". In: *Journal of the Operational Research Society* 66.11 [2015], pp. 1939–1947) differential equations are used to build a generic model of regime change. A detailed analysis of the stability of the system is included. The model offers some explanation of why oppressive regimes can follow an overthrow of a similarly oppressive regime: the underlying mathematical system is a stable cycle from which it is difficult to escape.

(James S Vandergraft. “A fluid flow model of networks of queues”. In: *Management Science* 29.10 [1983], pp. 1198–1208) uses differential equations as a framework for modelling queueing networks. This is interesting in its inception as differential equations are models for continuous quantities whereas queues are discrete type events (see Chapter ?? and 3 for more on this). The advantages of using differential equations are mainly in the computational efficiency.

The model presented in this chapter is deterministic: there is a single solution that remains the same. This is not always a precise model of reality: often systems are stochastic so that the inputs are not constant parameters but follow some random distribution. This is where stochastic differential equations are applied which is the subject of (Simo Särkkä and Arno Solin. *Applied stochastic differential equations*. Vol. 10. Cambridge University Press, 2019).



# 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  $\alpha$  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 represented in a stock and flow diagram as in Figure 5.1.

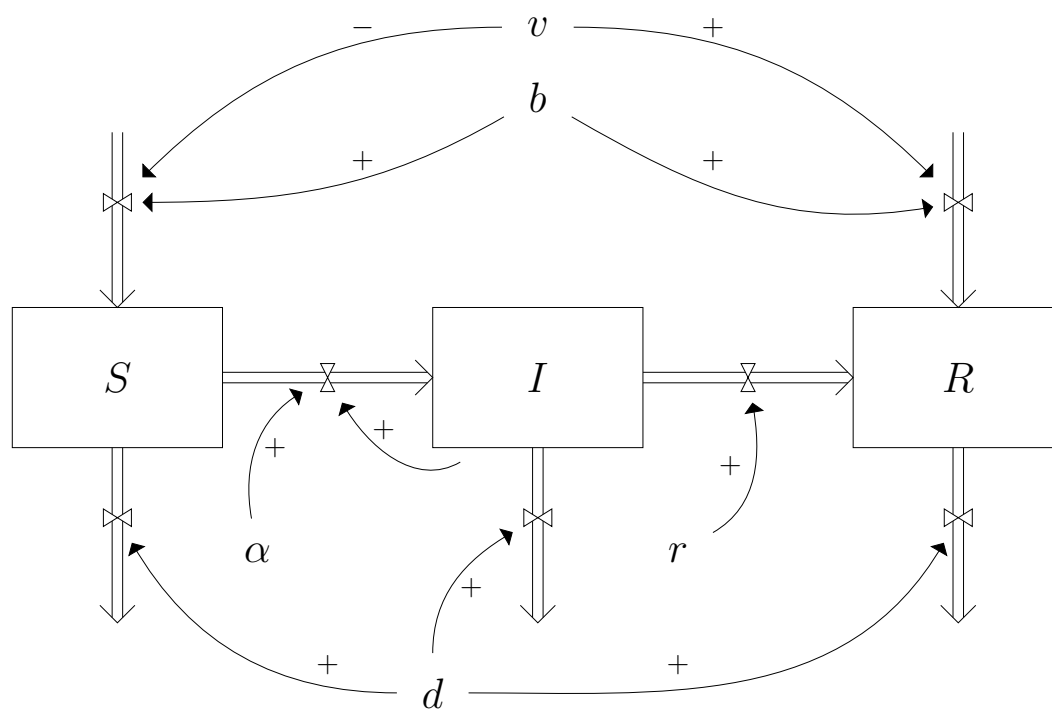


Figure 5.1 Diagrammatic representation of the epidemiology model



The system has three quantities, or ‘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 quantities or stocks are functions over time, and 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.

The behaviour of the quantities  $S$ ,  $I$  and  $R$  under these rules can be quantified by solving this system of differential equations. This system contains some non-linear terms, implying that this may be difficult to solve analytically, so a numerical method instead will be used.

A number of potential numerical methods to do this exist. The solvers that will be used in Python and R 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<sup>1</sup>, step through this sequence  $\{y_n\}$ , choosing appropriate values of  $h$  and employing other methods of error reduction.

### 5.3 SOLVING WITH PYTHON

---

Here the `odeint` method of the SciPy library will be used to numerically solve the above models.

First the system of differential equations described in Equations 5.1, 5.2 and 5.3 must be defined. This is done using a regular Python function, where the first two arguments are the system state and the current time respectively.

---

<sup>1</sup>These methods are studied in the area of Numerical Analysis. A good textbook is (Richard L Burden, J Douglas Faires, and Albert C Reynolds. *Numerical analysis*. Brooks/cole Pacific Grove, CA, 2001).

## Python input

```

651 def derivatives(y, t, vaccine_rate, birth_rate=0.01):
652     """Defines the system of differential equations that
653     describe the epidemiology model.
654
655     Args:
656         y: a tuple of three integers
657         t: a positive float
658         vaccine_rate: a positive float <= 1
659         birth_rate: a positive float <= 1
660
661     Returns:
662         A tuple containing dS, dI, and dR
663     """
664     infection_rate = 0.3
665     recovery_rate = 0.02
666     death_rate = 0.01
667     S, I, R = y
668     N = S + I + R
669     dSdt = (
670         -((infection_rate * S * I) / N)
671         + ((1 - vaccine_rate) * birth_rate * N)
672         - (death_rate * S)
673     )
674     dIdt = (
675         ((infection_rate * S * I) / N)
676         - (recovery_rate * I)
677         - (death_rate * I)
678     )
679     dRdt = (
680         (recovery_rate * I)
681         - (death_rate * R)
682         + (vaccine_rate * birth_rate * N)
683     )
684     return dSdt, dIdt, dRdt

```

Using this function returns the instantaneous rate of change for each of the three quantities,  $S$ ,  $I$  and  $R$ . Starting at time 0.0, with 4 susceptible individuals, 1 infected individual, 0 recovered individuals, and a vaccine rate of 50%, gives:

**Python input**

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

**Python output**

```
686 (-0.255, 0.21, 0.045)
```

this means that the number of susceptible individuals is expected 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. After a tiny fraction of a time unit these quantities will change, and thus the rates of change will change.

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

```

687 from scipy.integrate import odeint
688
689
690 def integrate_ode(
691     derivative_function,
692     t,
693     y0=(2999, 1, 0),
694     vaccine_rate=0.85,
695     birth_rate=0.01,
696 ):
697     """Numerically solve the system of differential equations.
698
699     Args:
700         derivative_function: a function returning a tuple
701                             of three floats
702         t: an array of increasing positive floats
703         y0: a tuple of three integers (default: (2999, 1, 0))
704         vaccine_rate: a positive float <= 1 (default: 0.85)
705         birth_rate: a positive float <= 1 (default: 0.01)
706
707     Returns:
708         A tuple of three arrays
709     """
710     results = odeint(
711         derivative_function,
712         y0,
713         t,
714         args=(vaccine_rate, birth_rate),
715     )
716     S, I, R = results.T
717     return S, I, R

```

This function can be used to investigate the difference in behaviour between a vaccination rate of 0% and a vaccination rate of 85%. The system will now be observed for two years, that is 730 days, in time steps of 0.01 days.

Begin with a vaccine rate of 0%:

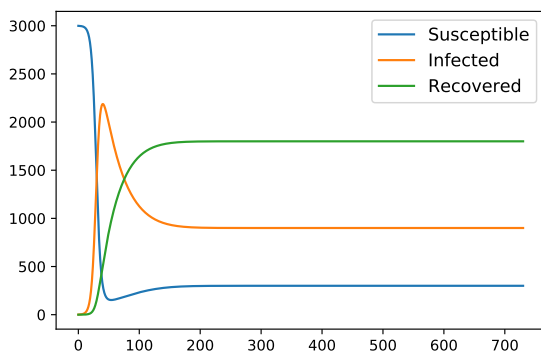


Figure 5.2 Output of code line 737-742

### Python input

```

718 import numpy as np
719 from scipy.integrate import odeint
720
721 t = np.arange(0, 730.01, 0.01)
722 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` a plot can be obtained to visualise their behaviour. The following code gives the plot shown in Figure 5.2.

### Python input

```

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

```

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 after some time period (around 300 time units) the levels of susceptible, infected, and recovered individuals stabilise, and

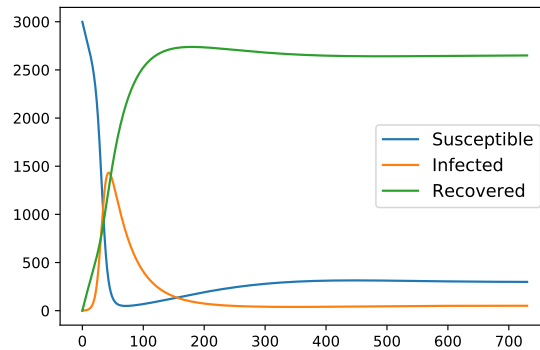


Figure 5.3 Output of code line 745-750

the disease becomes endemic. Once this 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

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

The following code gives the plot shown in Figure 5.3.

#### Python input

```
733 fig, ax = plt.subplots(1)
734 ax.plot(t, S, label='Susceptible')
735 ax.plot(t, I, label='Infected')
736 ax.plot(t, R, label='Recovered')
737 ax.legend(fontsize=12)
738 fig.savefig("plot_with_vaccine_python.pdf")
```

With vaccination the disease remains endemic, however 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.

This shows that vaccination lowers the percentage of the population living with the infection, which will lower the public healthcare system's costs. This saving will now be compared 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

```

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

```

Now the total daily cost with and without vaccination can be compared. Without vaccinations:

## Python input

```

771 cost = daily_cost(vaccine_rate=0.0)
772 print(round(cost, 2))

```

which gives



**Python output**

```
773 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**

```
774 cost = daily_cost(vaccine_rate=0.85)
775 print(round(cost, 2))
```

which gives

**Python output**

```
776 611903.36
```

So vaccinating 85% of the population 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

The `deSolve` library will be used to numerically solve the above epidemiology models.

First the system of differential equations described in Equations 5.1, 5.2 and 5.3 must be defined. This is done using an R function, where the arguments are the current time, system state and a list of other parameters.

## R input

```

777 #' Defines the system of differential equations that describe
778 #' the epidemiology model.
779 #'
780 #' @param t a positive float
781 #' @param y a tuple of three integers
782 #' @param vaccine_rate a positive float <= 1
783 #' @param birth_rate a positive float <= 1
784 #'
785 #' @return a list containing dS, dI, and dR
786 derivatives <- function(t, y, parameters){
787   infection_rate <- 0.3
788   recovery_rate <- 0.02
789   death_rate <- 0.01
790   with(as.list(c(y, parameters)), {
791     N <- S + I + R
792     dSdt <- ( - ( (infection_rate * S * I) / N) # nolint
793               + ( (1 - vaccine_rate) * birth_rate * N)
794               - (death_rate * S))
795     dIdt <- ( ( (infection_rate * S * I) / N) # nolint
796               - (recovery_rate * I)
797               - (death_rate * I))
798     dRdt <- ( (recovery_rate * I) # nolint
799               - (death_rate * R)
800               + (vaccine_rate * birth_rate * N))
801     list(c(dSdt, dIdt, dRdt)) # nolint
802   })
803 }

```

This function returns the instantaneous rate of change for each of the three quantities  $S$ ,  $I$  and  $R$ . Starting 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, gives:

## R input

```

804 derivatives(t = 0,
805             y = c(S = 4, I = 1, R = 0),
806             parameters = c(vaccine_rate = 0.5,
807                           birth_rate = 0.01)
808 )

```

**R output**

```

809 [[1]]
810 [1] -0.255  0.210  0.045

```

The number of susceptible individuals is expected 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. After a tiny fraction of a time unit these quantities will change, and thus the rates of change will change.

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**

```

811 library(deSolve) # nolint
812
813 #' Numerically solve the system of differential equations
814 #'
815 #' @param t an array of increasing positive floats
816 #' @param y0 list of integers (default: c(S=2999, I=1, R=0))
817 #' @param birth_rate a positive float <= 1 (default: 0.01)
818 #' @param vaccine_rate a positive float <= 1 (default: 0.85)
819 #'
820 #' @return a matrix of times, S, I and R values
821 integrate_ode <- function(times,
822                             y0 = c(S = 2999, I = 1, R = 0),
823                             birth_rate = 0.01,
824                             vaccine_rate = 0.84){
825   params <- c(birth_rate = birth_rate,
826               vaccine_rate = vaccine_rate)
827   ode(y = y0,
828       times = times,
829       func = derivatives,
830       parms = params)
831 }

```

This function can be used to investigate the difference in behaviour between a vaccination rate of 0% and a vaccination rate of 85%. The system will be observed for two years, that is 730 days, in time steps of 0.01 days.

Begin with a vaccine rate of 0%:

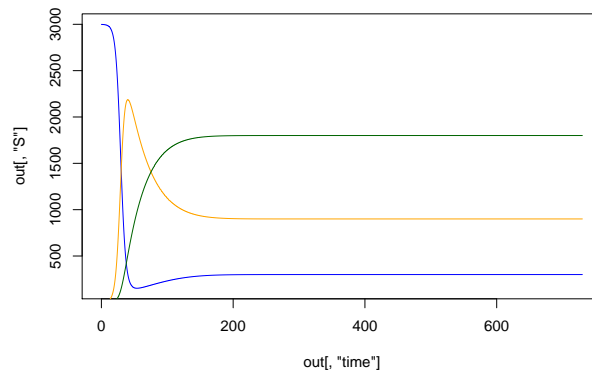


Figure 5.4 Output of code line 846-850

### R input

```
832 times <- seq(0, 730, by = 0.01)
833 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. These can be plotted to visualise their behaviour. The following code gives the plot shown in Figure 5.4.

### R input

```
834 pdf("plot_no_vaccine_R.pdf", width = 7, height = 5)
835 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
836 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
837 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
838 dev.off()
```

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 after some time period (around 300 time units) the levels of susceptible, infected, and recovered individuals stabilises, and the disease becomes endemic. 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%:

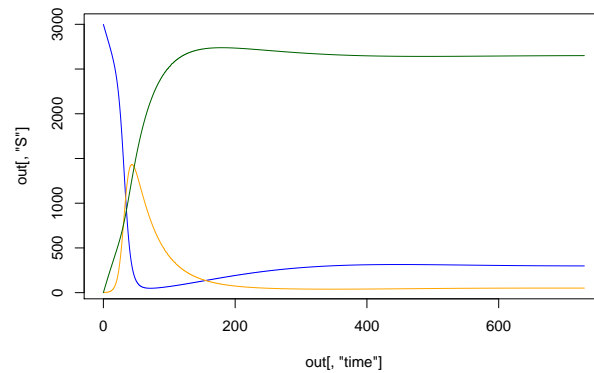


Figure 5.5 Output of code line 853-857

**R input**

```
839 times <- seq(0, 730, by = 0.01)
840 out <- integrate_ode(times, vaccine_rate = 0.85)
```

The following code gives the plot shown in Figure 5.5.

**R input**

```
841 pdf("plot_with_vaccine_R.pdf", width = 7, height = 5)
842 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
843 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
844 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
845 dev.off()
```

With vaccination the disease remains endemic, however 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.

This shows that vaccination lowers the percentage of the population living with the infection, which will lower the public healthcare system's costs. This saving will now be compared 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**

```

846 #' Calculates the daily cost to the public health
847 #' system after 2 years
848 #'
849 #' @param derivative_function: a function returning a
850 #'                             list of three floats
851 #' @param vaccine_rate: a positive float <= 1 (default: 0.85)
852 #'
853 #' @return the daily cost
854 daily_cost <- function(derivative_function = derivatives,
855                        vaccine_rate = 0.85){
856   max_time <- 730
857   time_step <- 0.01
858   birth_rate <- 0.01
859   vaccine_cost <- 220
860   medication_cost <- 10
861   times <- seq(0, max_time, by = time_step)
862   out <- integrate_ode(times, vaccine_rate = vaccine_rate)
863   N <- sum(tail(out[, c("S", "I", "R")], n = 1))
864   daily_vaccine_cost <- (N
865                        * birth_rate
866                        * vaccine_rate
867                        * vaccine_cost) / time_step
868   daily_medication_cost <- ( (tail(out[, "I"], n = 1)
869                          * medication_cost)) / time_step
870   daily_vaccine_cost + daily_medication_cost
871 }

```

The total daily cost with and without vaccination will now be compared. Without vaccinations:

**R input**

```

872 cost <- daily_cost(vaccine_rate = 0.0)
873 print(cost)

```

which gives

**R output**

```

874 [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

```
875 cost <- daily_cost(vaccine_rate = 0.85)
876 print(cost)
```

which gives

#### R output

```
877 [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 WIDER CONTEXT

System dynamics is an applied aspect of the more general mathematical field of dynamical systems. (Clark Robinson. *Dynamical systems: stability, symbolic dynamics, and chaos*. CRC press, 1998) gives a mathematical overview of the theory of dynamical systems and (Richard L Burden, J Douglas Faires, and Albert C Reynolds. *Numerical analysis*. Brooks/cole Pacific Grove, CA, 2001) is a good text on the numerical algorithms used to be able to observe the behaviour of these. For an overview of the type of application covered in this chapter see (J Martn Garca. *Theory and practical exercises of System Dynamics*. 2006).

Jay Forrester is recognised as the first person to use dynamical systems in the way shown in this chapter. His own account can be read in (Jay W Forrester. “The beginning of system dynamics”. In: [1995]). From Forrester’s initial application to industry in 1961 (Jay W Forrester. “Industrial dynamics. 1961”. In: *Pegasus Communications, Waltham, MA* [1961]) dynamical systems continue to be of use today in a wide range of areas. In (JM Coyle, D Exelby, and JSystemDynamicsInDefence-Analysis Holt. “System dynamics in defence analysis: some case studies”. In: *Journal of the Operational Research Society* 50.4 [1999], pp. 372–382) a case study of using dynamical systems for relevant modelling for the navy is described. (Jesús Isaac Vázquez-Serrano and RE Peimbert-Garca. “System dynamics applications in healthcare: A literature review”. In: *Proceedings of the international conference on industrial engineering and operations management*. 2020, pp. 10–12) gives a literature review of the application area of healthcare. For example, (Ian Cooper, Argha Mondal, and Chris G Antonopoulos. “A SIR model assumption for the spread of COVID-19 in different communities”. In: *Chaos, Solitons & Fractals* 139 [2020], p. 110057) applies the same model from this chapter to the study of the COVID pandemic.

In order to be able to fully capture all the relevant details of the system to

be modelled, an extension of system dynamics is to combine the methodology with discrete event simulation (see Chapter 3) in order to model discrete aspects of the system and/or agent based modelling (see Chapter 7) in order to observe emergent or learned behaviours. This is known as hybrid simulation, and an overview is given in (Brailsford et al., “Hybrid simulation modelling in operational research: A state-of-the-art review”). There are a number of ways of combining these methodologies, from comparison to full integration.



# IV

---

## Emergent Behaviour



# Game Theory

MOST of the time when modelling certain situations two approaches are valid: to make assumptions about the overall behaviour or to make assumptions about the detailed behaviour. The later can be thought of as 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, company A and company B, 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 for the companies are given in Table 6.2.

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. This would take the form of a fixed increase to the companies' profits,  $\epsilon$ , to be found, if they put on three taxis, shown in Table ??

From Table 6.2 it can be seen that if Company B chooses to use 3 vehicles while Company A chooses to only use 2 then Company B would get  $\frac{17}{20} + \epsilon$  and Company A would get  $\frac{1}{2}$  profits per hour. The question is: what value of  $\epsilon$  ensures both companies use 3 vehicles.

## 6.2 THEORY

In the case of this city, the interaction can be modelled using a mathematical object called a game, which here requires:

		Company B		
		1	2	3
Company A	1	1	$\frac{1}{2}$	$\frac{1}{3}$
	2	$\frac{3}{2}$	$\frac{19}{20}$	$\frac{1}{2}$
	3	$\frac{5}{3}$	$\frac{4}{5}$	$\frac{17}{20}$

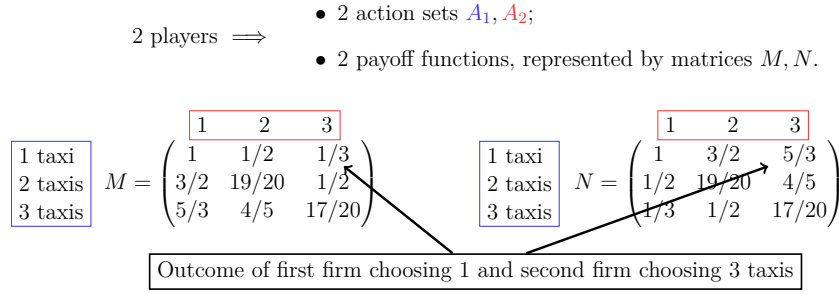
		Company B		
		1	2	3
Company A	1	1	$\frac{3}{2}$	$\frac{5}{3}$
	2	$\frac{1}{2}$	$\frac{19}{20}$	$\frac{4}{5}$
	3	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{17}{20}$

**Table 6.1** Profits (in GBP per hour) of each Taxi company based on the choice of vehicle number by all companies. The first table shows the profits for company A. The second table shows the profits for company B.

		Company B		
		1	2	3
Company A	1	1	$\frac{1}{2}$	$\frac{1}{3}$
	2	$\frac{3}{2}$	$\frac{19}{20}$	$\frac{1}{2}$
	3	$\frac{5}{3} + \epsilon$	$\frac{4}{5} + \epsilon$	$\frac{17}{20} + \epsilon$

		Company B		
		1	2	3
Company A	1	1	$\frac{3}{2}$	$\frac{5}{3} + \epsilon$
	2	$\frac{1}{2}$	$\frac{19}{20}$	$\frac{4}{5} + \epsilon$
	3	$\frac{1}{3}$	$\frac{1}{2}$	$\frac{17}{20} + \epsilon$

**Table 6.2** Profits (in GBP per hour) of each Taxi company based on the choice of vehicle number by all companies. The first table shows the profits for company A. The second table shows the profits for company B. The council’s financial incentive  $\epsilon$  is included.



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

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.

This is called a normal form game and is 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.

For this example, the two matrices would be:

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

A diagram of the system is shown in Figure 6.1

A strategy corresponds to a 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.

Both taxis always choosing to use 2 taxis (the second row/column) would correspond to the strategy:  $(0, 1, 0)$ . If both companies use this strategy and the row player

(who controls the rows) wants to improve their outcome it is 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 called a Nash equilibrium: when both players are playing a strategy that is the best response against the other.

An important fact is that a Nash equilibrium is guaranteed to exist. This was actually the theoretic result for which John Nash received a noble prize<sup>1</sup>. There are various algorithms that can be used for finding Nash equilibria, they involve a search through the pairs of spaces of possible strategies until pairs of best responses are found. Mathematical insight allows this to be done somewhat efficiently using algorithms that can be thought of as modifications of the algorithms used in linear programming. One such example is called the Lemke-Howson algorithm. A Nash equilibrium is not necessarily guaranteed to be arrived at through dynamic decision making. However, any stable behaviour that does emerge will be a Nash equilibrium, such emergent processes are the topics of evolutionary game theory<sup>2</sup>, learning algorithms<sup>3</sup> and/or agent based modelling which will be covered in Chapter 7.

### 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 and any offset. The Nashpy library will be used for this.

---

<sup>1</sup>John Nash proved the fact that any game has a Nash equilibrium in (John F Nash et al. “Equilibrium points in n-person games”. In: *Proceedings of the national academy of sciences* 36.1 [1950], pp. 48–49). Discussions and proofs for particular cases can be found in good Game Theory text books such as (Michael Maschler, Eilon Solan, and Shmuel Zamir. *Game theory*. Vol. 979. 2013, p. 4)

<sup>2</sup>Evolutionary game theory was formalised in (J Maynard Smith. “The theory of games and the evolution of animal conflicts”. In: *Journal of theoretical biology* 47.1 [1974], pp. 209–221) although some of the work of Robert Axelrod is evolutionary in principle (Robert Axelrod and William Donald Hamilton. “The evolution of cooperation”. In: *science* 211.4489 [1981], pp. 1390–1396)

<sup>3</sup>An excellent text on learning in games is (Drew Fudenberg et al. *The theory of learning in games*. Vol. 2. MIT press, 1998)

## Python input

```

878 import nashpy as nash
879 import numpy as np
880
881
882 def get_game(profits, offset=0):
883     """Return the game object with a given offset when 3 taxis
884     are provided.
885
886     Args:
887         profits: a matrix with expected profits
888         offset: a float
889
890     Returns:
891         A nashpy game object
892     """
893     new_profits = np.array(profits)
894     new_profits[2] += offset
895     return nash.Game(new_profits, new_profits.T)

```

This gives the game for the considered problem:

## Python input

```

896 import numpy as np
897
898 profits = np.array(
899     (
900         (1, 1 / 2, 1 / 3),
901         (3 / 2, 19 / 20, 1 / 2),
902         (5 / 3, 4 / 5, 17 / 20),
903     )
904 )
905 game = get_game(profits=profits)
906 print(game)

```

which gives:

### Python output

```

907 Bi matrix game with payoff matrices:
908
909 Row player:
910 [[1.          0.5          0.33333333]
911  [1.5         0.95         0.5         ]
912  [1.66666667 0.8          0.85         ]]
913
914 Column player:
915 [[1.          1.5          1.66666667]
916  [0.5         0.95         0.8         ]
917  [0.33333333 0.5          0.85         ]]

```

The function `get_equilibria` which will directly compute the equilibria:

### Python input

```

918 def get_equilibria(profits, offset=0):
919     """Return the equilibria for a given offset when 3 taxis
920     are provided.
921
922     Args:
923         profits: a matrix with expected profits
924         offset: a float
925
926     Returns:
927         A tuple of Nash equilibria
928     """
929     game = get_game(profits=profits, offset=offset)
930     return tuple(game.support_enumeration())

```

This can be used to obtain the equilibria in the game.

### Python input

```

931 equilibria = get_equilibria(profits=profits)

```

The equilibria are:



**Python input**

```

932 for eq in equilibria:
933     print(eq)

```

giving:

**Python output**

```

934 (array([0., 1., 0.]), array([0., 1., 0.]))
935 (array([0., 0., 1.]), array([0., 0., 1.]))
936 (array([0. , 0.7, 0.3]), array([0. , 0.7, 0.3]))

```

There are 3 Nash equilibria: 3 possible pairs of behaviour that the 2 companies could stabilise at:

- The first equilibrium  $((0, 1, 0), (0, 1, 0))$  corresponds to both firms always using 2 taxis;
- The second equilibrium  $((0, 0, 1), (0, 0, 1))$  corresponds to both firms always using 3 taxis;
- The third equilibrium  $((0, 0.7, 0.3), (0, 0.7, 0.3))$  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).

This can be used to find the number of Nash equilibria for a given offset and stop when there is a single equilibrium:

**Python input**

```

937 offset = 0
938 while len(get_equilibria(profits=profits, offset=offset)) > 1:
939     offset += 0.01

```

This gives a final offset value of:

**Python input**

```

940 print(round(offset, 2))

```

**Python output**

```
941 0.15
```

and now confirm that the Nash equilibrium is where both taxi firms provide three vehicles:

**Python input**

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

giving:

**Python output**

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

Therefore, in order to ensure that the maximum amount of taxis are used, the council should offer a £0.15 per hour incentive to both taxi companies for putting on 3 taxis.

## 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**

```

944 library(Recon)
945
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 = 0){
954   new_profits <- rbind(
955     profits[c(1, 2), ],
956     profits[3, ] + offset)
957   sim_nasheq(new_profits, t(new_profits))
958 }

```

This gives the pure Nash equilibria:

**R input**

```

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

```

which gives:

**R output**

```

966 $`Equilibrium 1`
967 [1] "2" "2"
968
969 $`Equilibrium 2`
970 [1] "3" "3"

```

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

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

There is in fact a third Nash equilibrium 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 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 gives the number of equilibria for a given offset and stops when there is a single equilibrium:

#### R input

```
971 offset <- 0
972 while (length(
973       get_equilibria(profits = profits, offset = offset)
974       ) > 1){
975   offset <- offset + 0.01
976 }
```

This gives a final offset value of:

#### R input

```
977 print(round(offset, 2))
```

#### R output

```
978 [1] 0.15
```

now confirm that the Nash equilibrium is where both taxi firms provide three vehicles:

#### R input

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

giving:

**R output**

```

980  `$Equilibrium 1`
981  [1] "3" "3"

```

Therefore, in order to ensure that the maximum amount of taxis are used, the council should offer a £0.15 per hour incentive to both taxi companies for putting on 3 taxis.

## 6.5 WIDER CONTEXT

The definition of a normal form game here as well as the solution concept of Nash equilibrium are common starting points for the use of game theory as a study of emergent behaviour. Other topics include different forms of games, for example extensive form which are represented by trees and more explicitly model asynchronous decision making. Other solution concepts involve the specific study of the emergence mechanisms through models based on natural evolutionary process: Moran processes or replicator dynamics. A good text book to read on these topics is (Michael Maschler, Eilon Solan, and Shmuel Zamir. *Game theory*. Vol. 979. 2013, p. 4).

John Nash whose life was portrayed in the 2001 movie “a beautiful mind” (which is an adaptation of (Sylvia Nasar. *A beautiful mind*. Simon and Schuster, 2011)) won the Noble prize for (John F Nash et al. “Equilibrium points in n-person games”. In: *Proceedings of the national academy of sciences* 36.1 [1950], pp. 48–49) in which he proved that a Nash equilibrium always exists. However, in (John Nash. “Non-cooperative games”. In: *Annals of mathematics* [1951], pp. 286–295) John Nash gives an application of Game Theory to a specific version of Poker.

Another application of the concept of Nash equilibrium is (Sarang Deo and Itai Gurvich. “Centralized vs. decentralized ambulance diversion: A network perspective”. In: *Management Science* 57.7 [2011], pp. 1300–1319) where the authors identify worst case scenarios for ambulance diversion: a practice where an emergency room will declare itself too full to accept new patients. When they are multiple emergency units serving a same population strategic behaviour becomes relevant. The authors of this paper identify the effect of this decentralised decision making and also propose an approach that is socially optimal: similarly to the Taxi problem considered in this chapter.

A specific area of a lot of research in Game theory is the study of cooperative behaviour. (Robert Axelrod and William Donald Hamilton. “The evolution of cooperation”. In: *science* 211.4489 [1981], pp. 1390–1396) started this work with his original computer tournaments with more recent work involving so called Zero-Determinant strategies which considered extortion as a mathematical concept (William H Press and Freeman J Dyson. “Iterated Prisoners Dilemma contains strategies that dominate any evolutionary opponent”. In: *Proceedings of the National Academy of Sciences* 109.26 [2012], pp. 10409–10413). A review and systemic analysis of some of the research on behaviour, of which game theory is a subset, is given in (Press and Dyson,

“Iterated Prisoners Dilemma contains strategies that dominate any evolutionary opponent”).

# Agent Based Simulation

SOMETIMES individual behaviours and interactions are well understood, and an understanding of how a whole population of such individuals might behave needed. For example psychologists and economists may know a lot about how individual spenders and vendors behave in response to given stimuli, but an understanding of how these stimuli might effect the macro-economy is necessary. Agent based simulation 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 categories of household, for example a household might be fans of Cardiff City FC or Swansea City AFC<sup>1</sup>. Each household has a preference for living close to households of the same kind, and will move around the city while their preferences are not satisfied. How will these individual preferences affect the overall distribution of fans in the city?

## 7.2 THEORY

The problem 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’,<sup>2</sup> 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 given utility function;

<sup>1</sup>Swansea and Cardiff are two cities in South Wales with rival football clubs.

<sup>2</sup>Thomas C Schelling. *Micromotives and macrobehavior*. WW Norton & Company, 2006.

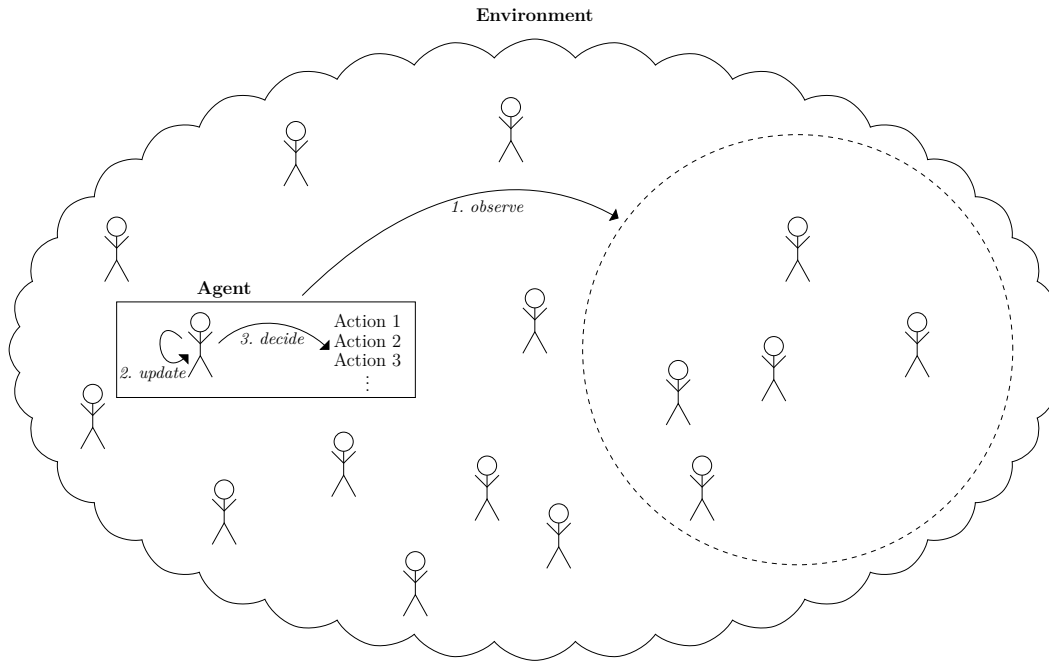


Figure 7.1 Representation of an agent interacting with its environment.

- An environment contains a number of agents and defines how their interactions affect each other. The agents may be homogeneous or heterogeneous, and the relationships 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 close to. 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, 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 or random and/or based on its own attributes from some learning process; with the ultimate aim of maximising its own utility. In practice, a utility can be represented by a function that maps the environment to some numeric value. This process happens to all agents in the environment, possibly simultaneously. This is summarised in Figure 7.1

For 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 them. Their choices of action are to move house or not to move house.

As a simplification the city will be modelled as a  $50 \times 50$  grid. Each cell of the grid is a house that can either contain a household of Cardiff City FC supporters, or contain a household of Swansea City AFC supporters. A house's neighbours are assumed to be the houses adjacent to it, horizontally, vertically, and diagonally. For



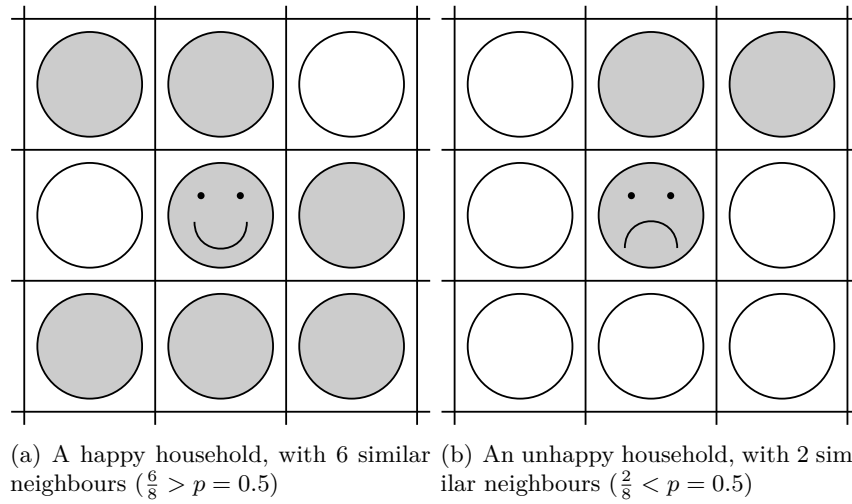


Figure 7.2 Example of a household happy and unhappy with its neighbours, when  $p = 0.5$ . Households supporting Cardiff City FC are shaded grey, households supporting Swansea City AFC are white.

mathematical simplicity, it is also assumed 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.

Every household has a preference  $p$ . This corresponds to the minimum proportion of neighbours they are happy to live 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.

The original problem stated that households 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. In fact, this can be simplified to consider the houses themselves as agents, who swap households with each other.

Therefore the 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 the overall structure of the city can be observed

from this agent based model, as it only explicitly defines individual behaviours and interactions. Any population level behaviour that may have emerged without explicit definition.

### 7.3 SOLVING WITH PYTHON

---

Agent based modelling lends itself well to a programming paradigm called object-orientated programming. This paradigm lets a number of *objects* from a set of instructions called a *class* to be built. These objects can both store information (in Python these are called *attributes*), and do things (in Python these are called *methods*). Object-orientated programming allow for the creation of new classes which can be used to implement the individual behaviours of an agent based model.

For this problem two classes will be built: a **House** and a **City** for them to live in.

The following libraries will be used:

#### Python input

```
982 import random
983 import itertools
984 import numpy as np
```

Now to define the **City**:

## Python input

```

985 class City:
986     def __init__(self, size, threshold):
987         """Initialises the City object.
988
989         Args:
990             size: an integer number of rows and columns
991             threshold: a number between 0 and 1 representing
992                 the minimum acceptable proportion of similar
993                 neighbours
994         """
995         self.size = size
996         sides = range(size)
997         self.coords = itertools.product(sides, sides)
998         self.houses = {
999             (x, y): House(x, y, threshold, self)
1000             for x, y in self.coords
1001         }
1002
1003     def run(self, n_steps):
1004         """Runs the simulation of a number of time steps.
1005
1006         Args:
1007             n_steps: an integer number of steps
1008         """
1009         for turn in range(n_steps):
1010             self.take_turn()
1011
1012     def take_turn(self):
1013         """Swaps all sad households."""
1014         sad = [h for h in self.houses.values() if h.sad()]
1015         random.shuffle(sad)
1016         i = 0
1017         while i <= len(sad) / 2:
1018             sad[i].swap(sad[-i])
1019             i += 1
1020
1021     def mean_satisfaction(self):
1022         """Finds the average household satisfaction.
1023
1024         Returns:
1025             The average city's household satisfaction
1026         """
1027         return np.mean(
1028             [h.satisfaction() for h in self.houses.values()]
1029         )

```

This defines a class, a template or a set of instructions that can be used to create instances of it, called objects. For the considered problem only one instance of the `City` class will be needed. 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 the `coords` are defined, a list of tuples representing all the possible coordinates of the grid, this uses the `itertools` library for efficient iteration.
- Finally `houses` is defined, a dictionary with grid coordinates as keys, and instances of the `House` class.

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 boundary 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

```

1030 class House:
1031     def __init__(self, x, y, threshold, city):
1032         """Initialises the House object.
1033
1034         Args:
1035             x: the integer x-coordinate
1036             y: the integer y-coordinate
1037             threshold: a number between 0 and 1 representing
1038                 the minimum acceptable proportion of similar
1039                 neighbours
1040             city: an instance of the City class
1041         """
1042         self.x = x
1043         self.y = y
1044         self.threshold = threshold
1045         self.kind = random.choice(["Cardiff", "Swansea"])
1046         self.city = city
1047
1048     def satisfaction(self):
1049         """Determines the household's satisfaction level.
1050
1051         Returns:
1052             A proportion
1053         """
1054         same = 0
1055         for x, y in itertools.product([-1, 0, 1], [-1, 0, 1]):
1056             ax = (self.x + x) % self.city.size
1057             ay = (self.y + y) % self.city.size
1058             same += self.city.houses[ax, ay].kind == self.kind
1059         return (same - 1) / 8
1060
1061     def sad(self):
1062         """Determines if the household is sad.
1063
1064         Returns:
1065             a Boolean
1066         """
1067         return self.satisfaction() < self.threshold
1068
1069     def swap(self, house):
1070         """Swaps two households.
1071
1072         Args:
1073             house: the house object to swap household with
1074         """
1075         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 if the household's satisfaction is below the minimum threshold.

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

A function to create and run one of these simulations will now be written with a given random seed, threshold, and number of steps. This function returns the resulting mean happiness:

### Python input

```

1076 def find_mean_happiness(seed, size, threshold, n_steps):
1077     """Create and run an instance of the simulation.
1078
1079     Args:
1080         seed: the random seed to use
1081         size: an integer number of rows and columns
1082         threshold: a number between 0 and 1 representing
1083             the minimum acceptable proportion of similar
1084             neighbours
1085         n_steps: an integer number of steps
1086
1087     Returns:
1088         The average city's household satisfaction after
1089         n_steps
1090     """
1091     random.seed(seed)
1092     C = City(size, threshold)
1093     C.run(n_steps)
1094     return C.mean_satisfaction()
```

Now consider each household with a threshold of 0.65, and compare the mean happiness after 0 steps and 100 steps. First 0 steps:

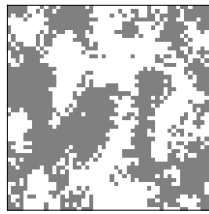
### Python input

```

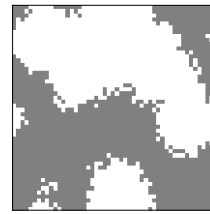
1095 print(find_mean_happiness(0, 50, 0.65, 0))
```



(a) At the beginning.



(b) After 20 time steps.



(c) After 100 time steps.

Figure 7.3 Plotted results from the Python code.

### Python output

1096 0.4998

This is well below the minimum threshold of 0.65, and so on average most households are unhappy. After 100 steps:

### Python input

1097 `print(find_mean_happiness(0, 50, 0.65, 100))`

### Python output

1098 0.9078

After 100 time steps the average satisfaction level is much higher. In fact, it is much higher than 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<sup>3</sup> 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 segregating over time.

## 7.4 SOLVING WITH R

Agent based modelling lends itself well to a programming paradigm called object-orientated programming. This paradigm lets a number of *objects* from a set of in-

<sup>3</sup>Schelling, *Micromotives and macrobehavior*.

structions called a *class* to be built. These objects can both store information (in the R library used here these are called *fields*), and do things (in the R library used here these are called *methods*). Object-orientated programming allow for the creation of new classes which can be used to implement the individual behaviours of an agent based model.

There are a number of ways of doing object-orientated programming in R. In this chapter, a package called **R6** will be used here.

For this problem two classes will be built: a **House** and a **City** for them to live in. Now to define the **City**<sup>4</sup>

---

<sup>4</sup>For the purposes of pagination, no documentation is included in the definition of the class.



## R input

```

1099 library(R6)
1100 city <- R6Class("City", list(
1101   size = NA,
1102   houses = NA,
1103   initialize = function(size, threshold) {
1104     self$size <- size
1105     self$houses <- c()
1106     for (x in 1:size) {
1107       row <- c()
1108       for (y in 1:size) {
1109         row <- c(row, house$new(x, y, threshold, self))
1110       }
1111       self$houses <- rbind(self$houses, row)
1112     } },
1113   run = function(n_steps) {
1114     if (n_steps > 0) {
1115       for (turn in 1:n_steps) {
1116         self$take_turn()
1117       } },
1118   take_turn = function() {
1119     sad <- c()
1120     for (house in self$houses) {
1121       if (house$sad()) {
1122         sad <- c(sad, house)
1123       } }
1124     sad <- sample(sad)
1125     num_sad <- length(sad)
1126     i <- 1
1127     while (i <= num_sad / 2) {
1128       sad[[i]]$swap(sad[[num_sad - i]])
1129       i <- i + 1
1130     } },
1131   mean_satisfaction = function() {
1132     mean(sapply(self$houses, function(x) x$satisfaction()))
1133   })
1134 )

```

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 the `houses` are 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, a list of all the houses with households that are unhappy with their neighbours is created; these are put in a random order and then working inwards from the boundary, 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 the emergent behaviour. This calculates the average satisfaction of all the houses in the grid.

In order to be able to create an instance of the above class, a `House` class is needed:

## R input

```

1135 house <- R6Class("House", list(
1136   x = NA,
1137   y = NA,
1138   threshold = NA,
1139   city = NA,
1140   kind = NA,
1141   initialize = function(x = NA,
1142                         y = NA,
1143                         threshold = NA,
1144                         city = NA) {
1145     self$x <- x
1146     self$y <- y
1147     self$threshold <- threshold
1148     self$city <- city
1149     self$kind <- sample(c("Cardiff", "Swansea"), 1)
1150   },
1151   satisfaction = function() {
1152     same <- 0
1153     for (x in -1:1) {
1154       for (y in -1:1) {
1155         ax <- ( (self$x + x - 1) %% self$city$size) + 1
1156         ay <- ( (self$y + y - 1) %% self$city$size) + 1
1157         if (self$city$houses[[ax, ay]]$kind == self$kind) {
1158           same <- same + 1
1159         } } }
1160     (same - 1) / 8
1161   },
1162   sad = function() {
1163     self$satisfaction() < self$threshold
1164   },
1165   swap = function(house) {
1166     old <- self$kind
1167     self$kind <- house$kind
1168     house$kind <- old
1169   })
1170 )

```

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

The `initialize` methods 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.

A function to create and run one of these simulations will now be written with a given random seed, threshold, and number of steps. This function return the resulting mean happiness:

### R input

```

1171 #' Create and run an instance of the simulation.
1172 #'
1173 #' @param seed: the random seed to use
1174 #' @param size: an integer number of rows and columns
1175 #' @param threshold: a number between 0 and 1 representing
1176 #'   the minimum acceptable proportion of similar neighbours
1177 #' @param n_steps: an integer number of steps
1178 #'
1179 #' @return The average city's household satisfaction
1180 #'   after n_steps
1181 find_mean_happiness <- function(seed, size,
1182                                threshold, n_steps){
1183   set.seed(seed)
1184   our_city <- city$new(size, threshold)
1185   our_city$run(n_steps)
1186   our_city$mean_satisfaction()
1187 }

```

Now consider each household with a threshold of 0.65, and compare the mean happiness after 0 steps and 100 steps. First 0 steps:

### R input

```

1188 print(find_mean_happiness(0, 50, 0.65, 0))

```

### R output

```

1189 [1] 0.4956

```

This is well below the minimum threshold of 0.65, and so on average most house-

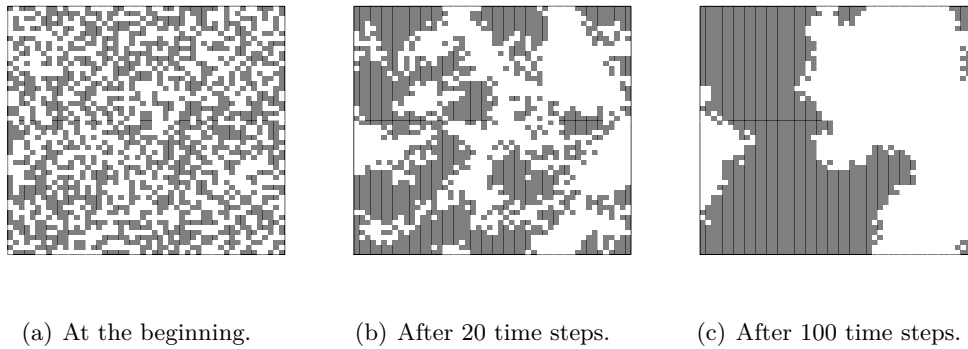


Figure 7.4 Plotted results from the R code.

holds are unhappy here. Let's run the simulation for 100 generations and see how this changes:

#### R input

```
1190 print(find_mean_happiness(0, 50, 0.65, 100))
```

#### R output

```
1191 [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,<sup>5</sup> 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 shows the households segregating over time.

## 7.5 WIDER CONTEXT

The simulations described in this chapter come under the larger umbrella term of multi agent systems, which discusses the theory of systems with multiple independent agents interacting with one another. A good source on the topic is (Yoav Shoham and

<sup>5</sup>Schelling, *Micromotives and macrobehavior*.

Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008).

The model described here is called Schelling's Model of Segregation, and is first described in (Thomas C Schelling. *Micromotives and macrobehavior*. WW Norton & Company, 2006). Another model considered as classic in this domain is a model of a flock of birds presented in (Craig W Reynolds. "Flocks, herds and schools: A distributed behavioral model". In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 1987, pp. 25–34), otherwise referred to as Boids, where the behaviours of flocks of birds are understood by capturing the individual interactions between individual birds. Conway's Game of Life, described in (Martin Gardner. "MATHEMATICAL GAMES: The fantastic combinations of John Conway's new solitaire game life". In: *Scientific American* 223.4 [1970], pp. 120–123) is another classic, which comes under the banner of cellular automata. Here cells on a grid either become alive or dead depending on a certain simplistic set of rules. Emergent behaviours observed due to these rules include self replicating as well as oscillating structures. In the 1970s agent based tournaments were held by Robert Axelrod (described in (Axelrod and Hamilton, "The evolution of cooperation")), which was the first of a number of studies using agent based modelling and game theory (see Chapter 6) to understand the emergence of cooperative behaviours.

In recent years, similar methodologies have been used in a variety of applications. (Diego A Daz, Ana Mara Jiménez, and Cristián Larroulet. "An agent-based model of school choice with information asymmetries". In: *Journal of Simulation* 15.1-2 [2021], pp. 130–147) models parents' choice of school, in (Iza Romanowska et al. "Agent-based modeling for archaeologists: Part 1 of 3". In: *Advances in archaeological practice* 7.2 [2019], pp. 178–184) archaeological population migration and trade dynamics are modelled, and (Peng Jing et al. "Agent-based simulation of autonomous vehicles: A systematic literature review". In: *IEEE Access* 8 [2020], pp. 79089–79103) offers a systematic literature review for the use of agent based modelling of autonomous vehicles.

# 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, using the least amount of time slots?

## 8.2 THEORY

Linear programming is a method that solves a type of optimisation problem of a number of variables by making use of some concepts of higher dimensional geometry.<sup>1</sup> Optimisation here refers to finding the variable that gives either the maximum or minimum of some linear function, called the objective function.

Linear programming employs algorithms such as the Simplex method to efficiently search some feasible convex region, stopping at the optimum. To do this, an objective function and constraints need to be defined.

To illustrate this a classic 2-dimensional example will be used: £50 of profit can be made on each tonne of paint A produced, and £60 profit on each tonne of paint B produced. A tonne of paint A needs 4 tonnes of component X and 5 tonnes of component Y. A tonne of paint B needs 6 tonnes of component X and 4 tonnes of component Y. Only 24 tonnes of X and 20 tonnes of Y are available per day. How much of paint A and paint B should be produced 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 components X and Y. They are written as:

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

Subject to:

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

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

Now this is a linear system in 2-dimensional space with coordinates A and B. These are called the decision variables, what is required are the values of A and B that optimises the objective function given by expression 8.1.

Inequalities 8.2 and 8.3 correspond to the amount of component X and Y available per day. These, along with the additional constraints that a negative amount of paint

<sup>1</sup>Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*. Vol. 271. Springer, 2014.

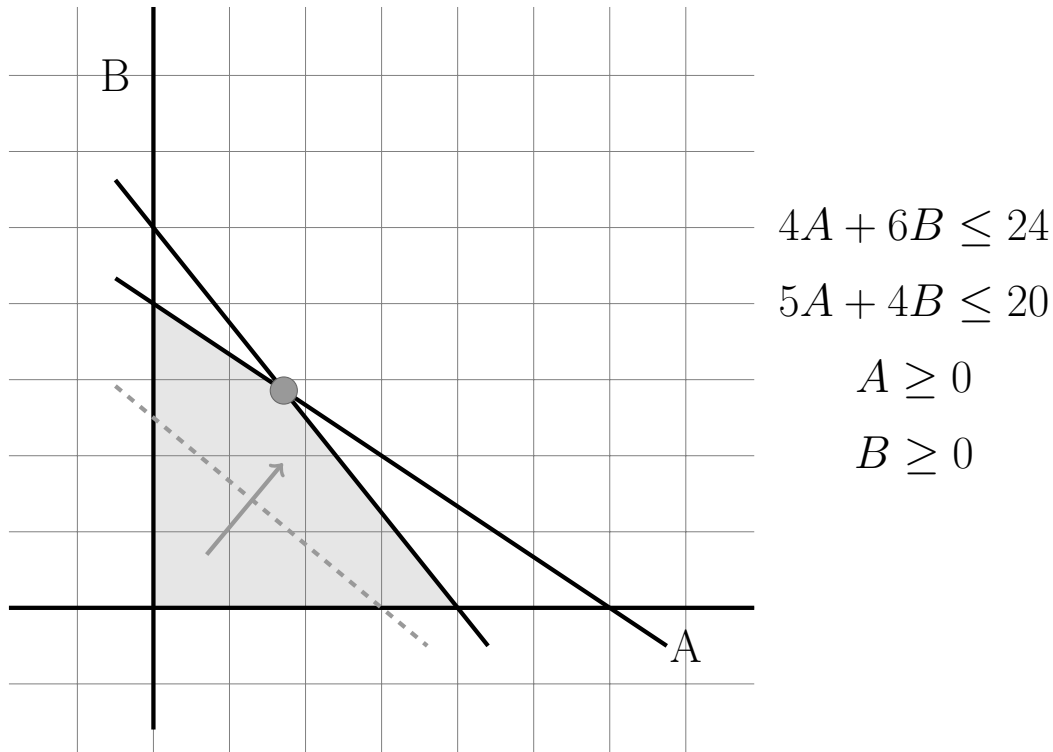


Figure 8.1 Visual representation of the paint linear program. The feasible convex region is shaded in grey; the objective function with arbitrary value is shown in a dashed line.

cannot be produced ( $A \geq 0$  and  $B \geq 0$ ), form a convex region, shown in Figure 8.1. This shaded region shows the pairs of values of  $A$  and  $B$  which are feasible, that is they satisfy the constraints.

Expression 8.1 corresponds to the total profit, which is the value to be maximised. As a line in 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. Some problems must be formulated as integer linear programs where the decision variables are restricted to integers. There are a number of methods that can help 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.<sup>2</sup>

<sup>2</sup>Conforti, Cornuéjols, Zambelli, et al., *Integer programming*.

Both Python and R have libraries that carry out the linear and integer programming algorithms. 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 which will now be formulated as an integer 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.

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.

Equation 8.6, ensures all modules are scheduled once and once only. Thus altogether the 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, the matrix  $X$  and the vector  $Y$  can be ‘flattened’ together to form this new variable. This is done by first ordering  $X$  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 is 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 the Python library Pulp will be used to formulate and solve the integer program. First a function to create the binary problem variables for a given set of times and modules is needed:

#### Python input

```

1192 import pulp
1193
1194
1195 def get_variables(modules, times):
1196     """Returns the binary variables for a given timetabling
1197     problem.
1198
1199     Args:
1200         modules: The complete collection of modules to be
1201                 timetabled.
1202         times: The collection of available time slots.
1203
1204     Returns:
1205         A tuple containing the decision variables x and y.
1206     """
1207     xshape = (modules, times)
1208     x = pulp.LpVariable.dicts("X", xshape, cat=pulp.LpBinary)
1209     y = pulp.LpVariable.dicts("Y", times, cat=pulp.LpBinary)
1210     return x, y

```

The specific modules and times relating to the problem can now be used to obtain the corresponding variables:

#### Python input

```

1211 Ac = [0, 1]
1212 Ao = [2, 3, 4]
1213 Bc = [5, 6]
1214 Bo = [7, 8]
1215 Cc = [9, 10]
1216 Co = [11, 12, 13]
1217 modules = Ac + Ao + Bc + Bo + Cc + Co
1218 times = range(14)
1219
1220 x, y = get_variables(modules=modules, times=times)

```

Now `y` is a dictionary of binary decision variables, with keys as elements of the list `times`.  $Y_3$  corresponds to the third day:

#### Python input

```
1221 print(y[3])
```

#### Python output

```
1222 Y_3
```

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

#### Python input

```
1223 print(x[2][5])
```

#### Python output

```
1224 X_2_5
```

The next step is to create a specific program with the corresponding variables, objective function, constraints and solve it. This is done with the following function:

## Python input

```

1225 def get_solution(Ac, Ao, Bc, Bo, Cc, Co, times):
1226     """Returns the binary variables corresponding to the
1227     solution of given timetabling problem.
1228
1229     Args:
1230         Ac: The set of core art modules
1231         Ao: The set of optional art modules
1232         Bc: The set of core biology modules
1233         Bo: The set of optional biology modules
1234         Cc: The set of core chemistry modules
1235         Co: The set of optional chemistry modules
1236         times: The collection of available time slots.
1237
1238     Returns:
1239         A tuple containing the decision variables x and y.
1240     """
1241     modules = Ac + Ao + Bc + Bo + Cc + Co
1242     x, y = get_variables(modules=modules, times=times)
1243
1244     prob = pulp.LpProblem("ExamScheduling", pulp.LpMinimize)
1245
1246     objective_function = sum([y[day] for day in times])
1247     prob += objective_function
1248
1249     M = 1 / len(modules)
1250     for day in times:
1251         prob += M * sum(x[m][day] for m in modules) <= y[day]
1252         prob += sum([x[mod][day] for mod in Ac + Ao]) <= 1
1253         prob += sum([x[mod][day] for mod in Bc + Bo + Co]) <= 1
1254         prob += sum([x[mod][day] for mod in Bc + Bo + Ac]) <= 1
1255         prob += sum([x[mod][day] for mod in Cc + Co + Bo]) <= 1
1256
1257     for mod in modules:
1258         prob += sum(x[mod][day] for day in times) == 1
1259
1260     prob.solve(pulp.apis.PULP_CBC_CMD(msg=False))
1261
1262     return x, y

```

Using this, the solution  $x$  of the original problem can be obtained:



**Python input**

```
1263 x, y = get_solution(  
1264     Ac=Ac, Ao=Ao, Bc=Bc, Bo=Bo, Cc=Cc, Co=Co, times=times  
1265 )
```

These can be inspected, for example  $x_{25}$  is a boolean variable relating to if module 2 is scheduled on the 5th day.

**Python input**

```
1266 print(x[2][5].value())
```

**Python output**

```
1267 0.0
```

This was assigned the value 0, and so module 2 was not scheduled for that day. However, module 2 was scheduled for day 9:

**Python input**

```
1268 print(x[2][9].value())
```

**Python output**

```
1269 1.0
```

This was assigned a value of 1, and so module 2 was scheduled for that day. The following function creates a readable schedule:

## Python input

```

1270 def get_schedule(x, y, Ac, Ao, Bc, Bo, Cc, Co, times):
1271     """Returns a human readable schedule corresponding to the
1272     solution of given timetabling problem.
1273
1274     Args:
1275         Ac: The set of core art modules
1276         Ao: The set of optional art modules
1277         Bc: The set of core biology modules
1278         Bo: The set of optional biology modules
1279         Cc: The set of core chemistry modules
1280         Co: The set of optional chemistry modules
1281         times: The collection of available time slots.
1282
1283     Returns:
1284         A string with the schedule
1285     """
1286     modules = Ac + Ao + Bc + Bo + Cc + Co
1287
1288     schedule = ""
1289     for day in times:
1290         if y[day].value() == 1:
1291             schedule += f"\nDay {day}: "
1292             for mod in modules:
1293                 if x[mod][day].value() == 1:
1294                     schedule += f"{mod}, "
1295     return schedule

```

Thus:

**Python input**

```

1296 schedule = get_schedule(
1297     x=x,
1298     y=y,
1299     times=times,
1300     Ac=Ac,
1301     Ao=Ao,
1302     Bc=Bc,
1303     Bo=Bo,
1304     Cc=Cc,
1305     Co=Co,
1306 )
1307 print(schedule)

```

gives:

**Python output**

```

1308 Day 0: 1, 12,
1309 Day 5: 0, 13,
1310 Day 6: 11,
1311 Day 7: 4, 6, 10,
1312 Day 8: 3, 5, 9,
1313 Day 9: 2, 7,
1314 Day 13: 8,

```

The order of the days do not matter here, but we 7 days are required in order to schedule all exams with no clashes, with at most three exams scheduled each day.

## 8.4 SOLVING WITH R

The R package ROI, the R Optimization Infrastructure will be used here. 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. The solver that will be used here is called the CBC MILP Solver, which needs to be installed. The `rcbc` package is also necessary but cannot be installed in the usual way. Installation instructions for both, depend on the operating system and can be found at the documentation page for the ROI package<sup>3</sup>.

The ROI package requires that the linear program is represented in its matrix form, with a one-dimensional array of decision variables. Therefore the form of the model described at the end of Section 9.2 will be used. Functions that define the

<sup>3</sup>As of the time of writing, this can be found at <https://roi.r-forge.r-project.org/installation.html>

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  $\star$  are needed.

First the objective function:

### R input

```
1315 #' Writes the row of coefficients for the objective function
1316 #'
1317 #' @param n_modules: the number of modules to schedule
1318 #' @param n_days: the maximum number of days to schedule
1319 #'
1320 #' @return the objective function row to minimise
1321 write_objective <- function(n_modules, n_days){
1322   all_days <- rep(0, n_modules * n_days)
1323   Ys <- rep(1, n_days)
1324   append(all_days, Ys)
1325 }
```

For 3 modules and 3 days:

### R input

```
1326 write_objective(n_modules = 3, n_days = 3)
```

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

### R output

```
1327 [1] 0 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**

```

1328 #' Writes the constraint row dealing with clashes
1329 #'
1330 #' @param clashes: a vector of module indices that all cannot
1331 #'                be scheduled at the same time
1332 #' @param day: an integer representing the day
1333 #'
1334 #' @return the constraint row corresponding to that set of
1335 #'         clashes on that day
1336 write_X_clashes <- function(clashes, day, n_days, n_modules){
1337   today <- rep(0, n_modules)
1338   today[clashes] = 1
1339   before_today <- rep(0, n_modules * (day - 1))
1340   after_today <- rep(0, n_modules * (n_days - day))
1341   all_days <- c(before_today, today, after_today)
1342   full_coeffs <- c(all_days, rep(0, n_days))
1343   full_coeffs
1344 }

```

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**

```

1345 #' Writes the constraint row to ensure that every module is
1346 #' scheduled once and only one
1347 #'
1348 #' @param module: an integer representing the module
1349 #'
1350 #' @return the constraint row corresponding to scheduling a
1351 #'         module on only one day
1352 write_X_requirements <- function(module, n_days, n_modules){
1353   today <- rep(0, n_modules)
1354   today[module] = 1
1355   all_days <- rep(today, n_days)
1356   full_coeffs <- c(all_days, rep(0, n_days))
1357   full_coeffs
1358 }

```

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

### R input

```

1359 #' Writes the constraint row representing the Y variable,
1360 #' whether at least one exam is scheduled on that day
1361 #'
1362 #' @param day: an integer representing the day
1363 #'
1364 #' @return the constraint row corresponding to creating Y
1365 write_Y_constraints <- function(day, n_days, n_modules){
1366   today <- rep(1, n_modules)
1367   before_today <- rep(0, n_modules * (day - 1))
1368   after_today <- rep(0, n_modules * (n_days - day))
1369   all_days <- c(before_today, today, after_today)
1370   all_Ys <- rep(0, n_days)
1371   all_Ys[day] = -n_modules
1372   full_coeffs <- append(all_days, all_Ys)
1373   full_coeffs
1374 }
```

Finally the following function uses all previous functions 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

```

1375 #' Writes all the constraints as a matrix, column of
1376 #' inequalities, and right hand side column.
1377 #'
1378 #' @param list_clashes: a list of vectors with sets of modules
1379 #'                       that cannot be scheduled at the same time
1380 #'
1381 #' @return f.con the LHS of the constraints as a matrix
1382 #' @return f.dir the directions of the inequalities
1383 #' @return f.rhs the values of the RHS of the inequalities
1384 write_constraints <- function(list_clashes, n_days, n_modules){
1385   all_rows <- c()
1386   all_dirs <- c()
1387   all_rhss <- c()
1388   n_rows <- 0
1389
1390   for (clash in list_clashes){
1391     for (day in 1:n_days){
1392       clashes <- write_X_clashes(clash, day, n_days, n_modules)
1393       all_rows <- append(all_rows, clashes)
1394       all_dirs <- append(all_dirs, "<=")
1395       all_rhss <- append(all_rhss, 1)
1396       n_rows <- n_rows + 1
1397     }
1398   }
1399
1400   for (module in 1:n_modules){
1401     reqs <- write_X_requirements(module, n_days, n_modules)
1402     all_rows <- append(all_rows, reqs)
1403     all_dirs <- append(all_dirs, "==")
1404     all_rhss <- append(all_rhss, 1)
1405     n_rows <- n_rows + 1
1406   }
1407
1408   for (day in 1:n_days){
1409     Yconstraints <- write_Y_constraints(day, n_days, n_modules)
1410     all_rows <- append(all_rows, Yconstraints)
1411     all_dirs <- append(all_dirs, "<=")
1412     all_rhss <- append(all_rhss, 0)
1413     n_rows <- n_rows + 1
1414   }
1415
1416   f.con <- matrix(all_rows, nrow = n_rows, byrow = TRUE)
1417   f.dir <- all_dirs
1418   f.rhs <- all_rhss
1419   list(f.con, f.dir, f.rhs)
1420 }

```

For demonstration, with 2 modules and 2 possible days, with the single constraint that both modules cannot be scheduled at the same time, then:

### R input

```
1421 write_constraints(list_clashes = list(c(1, 2)),
1422                   n_days = 2,
1423                   n_modules = 2)
```

This would give 3 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

```
1424 [[1]]
1425      [,1] [,2] [,3] [,4] [,5] [,6]
1426 [1,]    1    1    0    0    0    0
1427 [2,]    0    0    1    1    0    0
1428 [3,]    1    0    1    0    0    0
1429 [4,]    0    1    0    1    0    0
1430 [5,]    1    1    0    0   -2    0
1431 [6,]    0    0    1    1    0   -2
1432
1433 [[2]]
1434 [1] "<=" "<=" "==" "==" "<=" "<="
1435
1436 [[3]]
1437 [1] 1 1 1 1 0 0
```

Now, the problem will be solved. First some parameters, including the sets of modules that all share students, that is the list of clashes are needed:



**R input**

```

1438 n_modules = 14
1439 n_days = 14
1440
1441 Ac <- c(0, 1)
1442 Ao <- c(2, 3, 4)
1443 Bc <- c(5, 6)
1444 Bo <- c(7, 8)
1445 Cc <- c(9, 10)
1446 Co <- c(11, 12, 13)
1447
1448 list_clashes <- list(
1449   c(Ac, Ao),
1450   c(Bc, Bo, Co),
1451   c(Bc, Bo, Ac),
1452   c(Bo, Cc, Co)
1453 )

```

Then, the functions defined above are used to create the objective function and the 3 elements of the constraints:

**R input**

```

1454 constraints <- write_constraints(list_clashes = list_clashes,
1455                                n_days = n_days,
1456                                n_modules = n_modules)
1457 f.con <- constraints[[1]]
1458 f.dir <- constraints[[2]]
1459 f.rhs <- constraints[[3]]
1460 f.obj <- write_objective(n_modules = n_modules, n_days = n_days)

```

Finally, once these objects are in place, the ROI library is used to construct an optimisation problem object:

Now to solve:

```
R input
1469 sol <- ROI_solve(milp)
```

```
R input  
1470 print(sol$solution)
```

	R output
1471	[1] 0 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
1472	[30] 0
1473	[59] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
1474	[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 0
1475	[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 0 1 0 0
1476	[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 0
1477	[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
1478	[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**

```

1479 #' Gives a human readable schedule corresponding to the
1480 #' solution of a given timetable problem.
1481 #'
1482 #' @param sol: a solution to the timetabling problem
1483 #' @param n_modules: the number of modules to schedule
1484 #' @param n_days: the maximum number of days to schedule
1485 #'
1486 #' @return A string with the schedule
1487 get_schedule <- function(sol, n_days, n_modules){
1488   schedule = ""
1489   for (day in 1:n_days){
1490     if (sol$solution[(n_days * n_modules) + day] == 1){
1491       schedule <- paste(schedule, "\n", "Day", day, ":")
1492       for (module in 1:n_modules){
1493         var <- ((day - 1) * n_modules) + module
1494         if (sol$solution[var] == 1){
1495           schedule <- paste(schedule, module)
1496         }
1497       }
1498     }
1499   }
1500   schedule
1501 }

```

Thus:

**R input**

```

1502 schedule <- get_schedule(
1503   sol = sol,
1504   n_days = n_days,
1505   n_modules = n_modules
1506 )
1507 cat(schedule)

```

gives:

**R output**

```

1508 "Day 2 : 4 11"
1509 "Day 6 : 1 12"
1510 "Day 8 : 7"
1511 "Day 10 : 8"
1512 "Day 11 : 3 13"
1513 "Day 12 : 2 6 9 14"
1514 "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 WIDER CONTEXT

The overview given here on linear programming covers a wide breath of the subject although not much depth. For specific algorithmic approaches to the underlying algorithms and problem types, such as branch and bound and cutting plane methods as well as some minor extensions see (Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*. Vol. 271. Springer, 2014; Alan Sultan. *Linear programming: An introduction with applications*. Elsevier, 2014).

The efficiency of a linear programme as well as the ability to model linear situations imply that it is often used for a variety of applications. Theatre scheduling as one such application is given in (Francesca Guerriero and Rosita Guido. “Operational research in the management of the operating theatre: a survey”. In: *Health care management science* 14.1 [2011], pp. 89–114). However, scheduling is a wide ranging sub application of linear programming which can also be used to schedule sport seasons (Guillermo Durán et al. “Scheduling the Chilean soccer league by integer programming”. In: *Interfaces* 37.6 [2007], pp. 539–552).

Other applications include the transportation problem (Ocotlán Daz-Parra et al. “A survey of transportation problems”. In: *Journal of Applied Mathematics* 2014 [2014]) which can be used to find a best allocation of a fleet of delivery vehicles; fire station location problem (JAM Schreuder. “Application of a location model to fire stations in Rotterdam”. In: *European Journal of Operational Research* 6.2 [1981], pp. 212–219) used to minimise travel times to at-risk areas; and the bin packing problem (Mhand Hifi et al. “A linear programming approach for the three-dimensional bin-packing problem”. In: *Electronic Notes in Discrete Mathematics* 36 [2010], pp. 993–1000) in which a number of, possibly irregular, shapes are packed into the smallest possible number of bins.

# 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

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

The stops are drawn in Figure 9.2.

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

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

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

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

## 9.2 THEORY

This problem is called a travelling salesman problem, which can often be inefficient to solve using exact methods.<sup>1</sup> Heuristics are a family of methods that can be used to

<sup>1</sup>Zbigniew Michalewicz and David B Fogel. *How to solve it: modern heuristics*. Springer Science & Business Media, 2013.

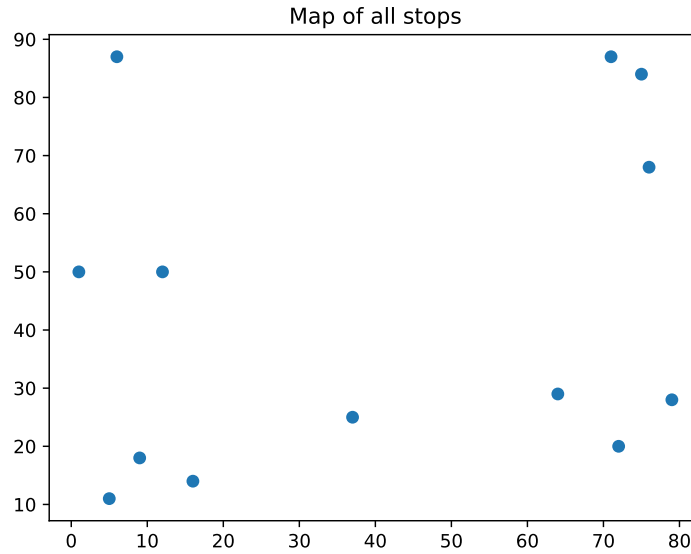


Figure 9.1 The positions of the required stops.

find a *sufficiently good* solution, though not necessarily the optimal solution, where the emphasis is on prioritising computational efficiency.

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

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

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

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

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

Thus, with:

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

We have:

$$\begin{aligned} C((1, 2, 3, 1)) &= d_{12} + d_{23} + d_{31} = 1 + 15 + 3 = 19 \\ C((1, 3, 2, 1)) &= d_{13} + d_{32} + d_{21} = 3 + 3 + 1 = 7 \end{aligned}$$

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  $\tilde{t}^{(1)} \in N(t)$  is obtained by swapping the 2nd and 5th stops.

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

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

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

Examples of these tours are shown in Figure 9.3.

### 9.3 SOLVING WITH PYTHON

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

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

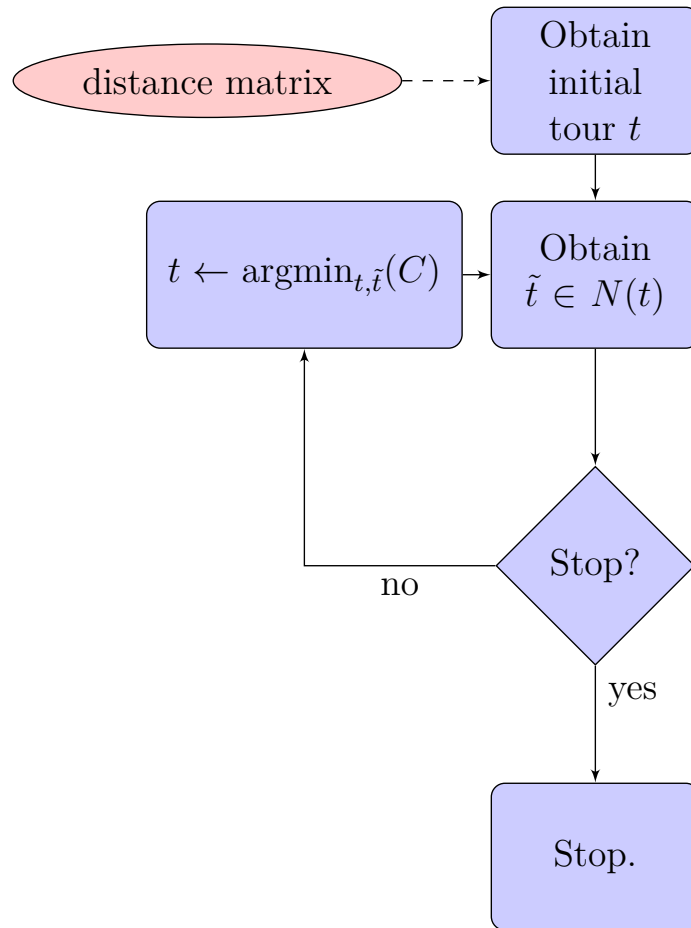


Figure 9.2 The general neighbourhood search algorithm.  $N(t)$  refers to some neighbourhood of  $t$ .

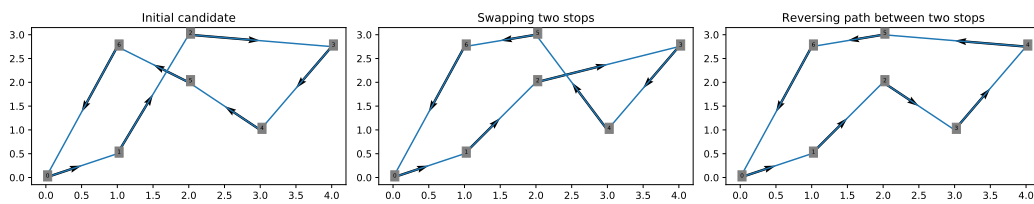


Figure 9.3 The effect of two neighbourhood operators on  $t$ .  $\tilde{t}^{(1)}$  is obtained by swapping stops 3 and 5.  $\tilde{t}^{(2)}$  is obtained by reversing the path between stops 2 and 5.



## Python input

```

1515 import numpy as np
1516
1517
1518 def get_initial_candidate(number_of_stops, seed):
1519     """Return an random initial tour.
1520
1521     Args:
1522         number_of_stops: The number of stops
1523         seed: An integer seed.
1524
1525     Returns:
1526         A tour starting an ending at stop with index 0.
1527     """
1528     internal_stops = list(range(1, number_of_stops))
1529     np.random.seed(seed)
1530     np.random.shuffle(internal_stops)
1531     return [0] + internal_stops + [0]

```

This gives a random tour on 13 stops:

## Python input

```

1532 number_of_stops = 13
1533 seed = 0
1534 initial_candidate = get_initial_candidate(
1535     number_of_stops=number_of_stops,
1536     seed=seed,
1537 )
1538 print(initial_candidate)

```

## Python output

```

1539 [0, 7, 12, 5, 11, 3, 9, 2, 8, 10, 4, 1, 6, 0]

```

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

## Python input

```

1540 def get_cost(tour, distance_matrix):
1541     """Return the cost of a tour.
1542
1543     Args:
1544         tour: A given tuple of successive stops.
1545         distance_matrix: The distance matrix of the problem.
1546
1547     Returns:
1548         The cost
1549     """
1550     return sum(
1551         distance_matrix[current_stop, next_stop]
1552         for current_stop, next_stop in zip(tour[:-1], tour[1:])
1553     )

```

## Python input

```

1554 distance_matrix = np.array(
1555     (
1556         (0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1557         (35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1558         (35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1559         (29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1560         (70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1561         (35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1562         (42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1563         (27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1564         (24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1565         (44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1566         (58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1567         (71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1568         (69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0),
1569     )
1570 )
1571 cost = get_cost(
1572     tour=initial_candidate,
1573     distance_matrix=distance_matrix,
1574 )
1575 print(cost)

```

## Python output

```
1576 827
```

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

## Python input

```
1577 def swap_stops(tour):
1578     """Return a new tour by swapping two stops.
1579
1580     Args:
1581         tour: A given tuple of successive stops.
1582
1583     Returns:
1584         A tour
1585     """
1586     number_of_stops = len(tour) - 1
1587     i, j = np.random.choice(range(1, number_of_stops), 2)
1588     new_tour = list(tour)
1589     new_tour[i], new_tour[j] = tour[j], tour[i]
1590     return new_tour
```

Applying this neighbourhood operator to the initial candidate gives:

## Python input

```
1591 print(swap_stops(initial_candidate))
```

which swaps the 10th and 12th stops:

## Python output

```
1592 [0, 7, 12, 5, 11, 3, 9, 2, 8, 1, 4, 10, 6, 0]
```

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

## Python input

```

1593 def run_neighbourhood_search(
1594     distance_matrix,
1595     iterations,
1596     seed,
1597     neighbourhood_operator=swap_stops,
1598 ):
1599     """Returns a tour by carrying out a neighbourhood search.
1600
1601     Args:
1602         distance_matrix: the distance matrix
1603         iterations: the number of iterations for which to
1604                     run the algorithm
1605         seed: a random seed
1606         neighbourhood_operator: the neighbourhood operator
1607                                (default: swap_stops)
1608
1609     Returns:
1610         A tour
1611     """
1612     number_of_stops = len(distance_matrix)
1613     candidate = get_initial_candidate(
1614         number_of_stops=number_of_stops,
1615         seed=seed,
1616     )
1617
1618     best_cost = get_cost(
1619         tour=candidate,
1620         distance_matrix=distance_matrix,
1621     )
1622
1623     for _ in range(iterations):
1624         new_candidate = neighbourhood_operator(candidate)
1625         if (
1626             cost := get_cost(
1627                 tour=new_candidate,
1628                 distance_matrix=distance_matrix,
1629             )
1630         ) <= best_cost:
1631             best_cost = cost
1632             candidate = new_candidate
1633
1634     return candidate

```

Now running this for 1000 iterations:

### Python input

```
1635 number_of_iterations = 1000
1636
1637 solution_with_swap_stops = run_neighbourhood_search(
1638     distance_matrix=distance_matrix,
1639     iterations=number_of_iterations,
1640     seed=seed,
1641     neighbourhood_operator=swap_stops,
1642 )
1643 print(solution_with_swap_stops)
```

gives:

### Python output

```
1644 [0, 7, 2, 8, 5, 3, 1, 9, 12, 11, 4, 10, 6, 0]
```

This has a cost:

### Python input

```
1645 cost = get_cost(
1646     tour=solution_with_swap_stops,
1647     distance_matrix=distance_matrix,
1648 )
1649 print(cost)
```

### Python output

```
1650 362
```

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

It is important to note that this may not be the optimal route, and different algorithms may produce better solutions. For example, one way to modify the algorithm is to use a different neighbourhood operator. Instead of swapping two stops, reverse the path between those two stops. The `reverse_path` function does this:

## Python input

```

1651 def reverse_path(tour):
1652     """Return a new tour by reversing the path between two
1653     stops.
1654
1655     Args:
1656         tour: A given tuple of successive stops.
1657
1658     Returns:
1659         A tour
1660     """
1661     number_of_stops = len(tour) - 1
1662     stops = np.random.choice(range(1, number_of_stops), 2)
1663     i, j = sorted(stops)
1664     new_tour = tour[:i] + tour[i : j + 1][::-1] + tour[j + 1 :]
1665     return new_tour

```

Applying this neighbourhood operator to the initial candidate gives:

## Python input

```

1666 print(reverse_path(initial_candidate))

```

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

## Python output

```

1667 [0, 7, 4, 10, 8, 2, 9, 3, 11, 5, 12, 1, 6, 0]

```

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

<sup>2</sup>The 2 opt algorithm was first published in (Georges A Croes. “A method for solving traveling-salesman problems”. In: *Operations research* 6.6 [1958], pp. 791–812).

**Python input**

```

1668 solution_with_reverse_path = run_neighbourhood_search(
1669     distance_matrix=distance_matrix,
1670     iterations=number_of_iterations,
1671     seed=seed,
1672     neighbourhood_operator=reverse_path,
1673 )
1674 print(solution_with_reverse_path)

```

gives:

**Python output**

```

1675 [0, 8, 5, 3, 1, 9, 12, 11, 4, 10, 6, 2, 7, 0]

```

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

**Python input**

```

1676 cost = get_cost(
1677     tour=solution_with_reverse_path,
1678     distance_matrix=distance_matrix,
1679 )
1680 print(cost)

```

which gives:

**Python output**

```

1681 299

```

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

## 9.4 SOLVING WITH R

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

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

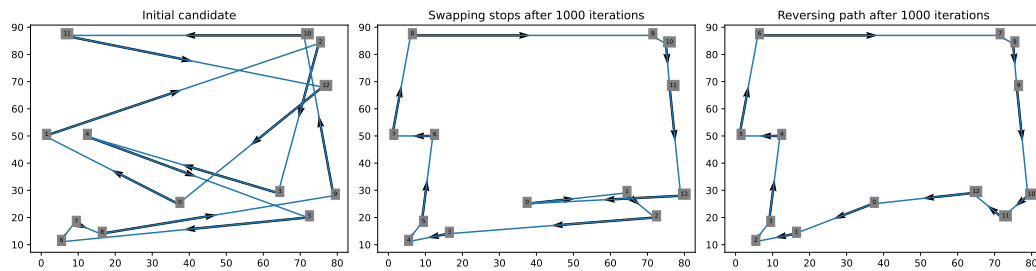


Figure 9.4 The final tours obtained by using the neighbourhood search in Python.

### R input

```

1682 #' Return an random initial tour.
1683 #'
1684 #' @param number_of_stops The number of stops.
1685 #' @param seed An integer seed.
1686 #'
1687 #' @return A tour starting an ending at stop with index 0.
1688 get_initial_candidate <- function(number_of_stops, seed){
1689   internal_stops <- 1:(number_of_stops - 1)
1690   set.seed(seed)
1691   internal_stops <- sample(internal_stops)
1692   c(0, internal_stops, 0)
1693 }

```

This gives a random tour on 13 stops:

### R input

```

1694 number_of_stops <- 13
1695 seed <- 1
1696 initial_candidate <- get_initial_candidate(
1697   number_of_stops = number_of_stops,
1698   seed = seed)
1699 print(initial_candidate)

```

### R output

```

1700 [1] 0 9 4 7 1 2 5 3 8 6 11 12 10 0

```



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

### R input

```

1701 #' Return the cost of a tour
1702 #'
1703 #' @param tour A given vector of successive stops.
1704 #' @param seed The distance matrix of the problem.
1705 #'
1706 #' @return The cost
1707 get_cost <- function(tour, distance_matrix){
1708     pairs <- cbind(tour[-length(tour)], tour[-1]) + 1
1709     sum(distance_matrix[pairs])
1710 }

```

### R input

```

1711 distance_matrix <- rbind(
1712     c(0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1713     c(35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1714     c(35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1715     c(29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1716     c(70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1717     c(35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1718     c(42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1719     c(27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1720     c(24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1721     c(44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1722     c(58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1723     c(71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1724     c(69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0)
1725 )
1726 cost <- get_cost(
1727     tour = initial_candidate,
1728     distance_matrix = distance_matrix)
1729 print(cost)

```

### R output

```

1730 [1] 709

```

Now a function for a neighbourhood operator will be written, `swap_stops`: swapping two stops in a given tour.

### R input

```
1731 #' Return a new tour by swapping two stops.
1732 #'
1733 #' @param tour A given vector of successive stops.
1734 #'
1735 #' @return A tour
1736 swap_stops <- function(tour){
1737   number_of_stops <- length(tour) - 1
1738   stops_to_swap <- sample(2:number_of_stops, 2)
1739   new_tour <- replace(x = tour,
1740                      list = stops_to_swap,
1741                      values = rev(tour[stops_to_swap]))
1742 }
```

Applying this neighbourhood operator to the initial candidate gives:

### R input

```
1743 print(swap_stops(initial_candidate))
```

which swaps the 6th and 11th stops:

### R output

```
1744 [1] 0 9 4 7 1 11 5 3 8 6 2 12 10 0
```

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

## R input

```

1745 #' Returns a tour by carrying out a neighbourhood search
1746 #'
1747 #' @param distance_matrix: the distance matrix
1748 #' @param iterations: the number of iterations for
1749 #'                      which to run the algorithm
1750 #' @param seed: a random seed (default: None)
1751 #' @param neighbourhood_operator: the neighbourhood operation
1752 #'                                (default: swap_stops)
1753 #'
1754 #' @return A tour
1755 run_neighbourhood_search <- function(
1756   distance_matrix,
1757   iterations,
1758   seed = NA,
1759   neighbourhood_operator = swap_stops
1760 ){
1761   number_of_stops <- nrow(distance_matrix)
1762   candidate <- get_initial_candidate(
1763     number_of_stops = number_of_stops,
1764     seed = seed
1765   )
1766
1767   best_cost <- get_cost(
1768     tour = candidate,
1769     distance_matrix = distance_matrix
1770   )
1771
1772   for (repetition in 1:iterations) {
1773     new_candidate <- neighbourhood_operator(candidate)
1774     cost <- get_cost(
1775       tour = new_candidate,
1776       distance_matrix = distance_matrix
1777     )
1778     if (cost <= best_cost) {
1779       best_cost <- cost
1780       candidate <- new_candidate
1781     }
1782   }
1783   candidate
1784 }
1785

```

Now running this for 1000 iterations:

### R input

```
1786 number_of_iterations <- 1000
1787 solution_with_swap_stops <- run_neighbourhood_search(
1788     distance_matrix = distance_matrix,
1789     iterations = number_of_iterations,
1790     seed = seed,
1791     neighbourhood_operator = swap_stops
1792 )
1793 print(solution_with_swap_stops)
```

gives:

### R output

```
1794 [1] 0 11 4 10 6 2 7 12 9 1 3 5 8 0
```

This has a cost:

### R input

```
1795 cost <- get_cost(
1796     tour = solution_with_swap_stops,
1797     distance_matrix = distance_matrix
1798 )
1799 print(cost)
```

which gives:

### R output

```
1800 [1] 360
```

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

It is important to note that this may not be the optimal route, and different algorithms may produce better solutions. For example, one way to modify the algorithm is to use a different neighbourhood operator. Instead of swapping two stops, reverse the path between those two stops. The `reverse_path` function does this:

**R input**

```

1801 #' Return a new tour by reversing the path between two stops.
1802 #'
1803 #' @param tour A given vector of successive stops.
1804 #'
1805 #' @return A tour
1806 reverse_path <- function(tour){
1807   number_of_stops <- length(tour) - 1
1808   stops_to_swap <- sample(2:number_of_stops, 2)
1809   i <- min(stops_to_swap)
1810   j <- max(stops_to_swap)
1811   new_order <- c(
1812     c(1: (i - 1)),
1813     c(j:i),
1814     c( (j + 1): length(tour))
1815   )
1816   tour[new_order]
1817 }

```

Applying this neighbourhood operator to the initial candidate gives:

**R input**

```

1818 print(reverse_path(initial_candidate))

```

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

**R output**

```

1819 [1] 0 9 10 12 11 6 8 3 5 2 1 7 4 0

```

Now running the neighbourhood search for 1000 iterations using the `reverse_path` neighbourhood operator, which corresponds to an algorithm called the “2 opt” algorithm<sup>3t</sup>

<sup>3t</sup>The 2 opt algorithm was first published in (Croes, “A method for solving traveling-salesman problems”).

**R input**

```

1820 number_of_iterations <- 1000
1821 solution_with_reverse_path <- run_neighbourhood_search(
1822     distance_matrix = distance_matrix,
1823     iterations = number_of_iterations,
1824     seed = seed,
1825     neighbourhood_operator = reverse_path
1826 )
1827 print(solution_with_reverse_path)

```

gives:

**R output**

```

1828 [1] 0 7 2 6 10 4 11 12 9 1 3 5 8 0

```

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

**R input**

```

1829 cost <- get_cost(
1830     tour = solution_with_reverse_path,
1831     distance_matrix = distance_matrix
1832 )
1833 print(cost)

```

which gives:

**R output**

```

1834 [1] 299

```

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

## 9.5 WIDER CONTEXT

Heuristic methods, sometimes referred to as meta-heuristics, are a whole family of algorithms used to find approximate solutions to combinatorial optimisation problems. An overview is given in (Omid Bozorg-Haddad, Mohammad Solgi, and Hugo A Loáigiga. *Meta-heuristic and evolutionary algorithms for engineering optimization*. John Wiley & Sons, 2017). These algorithms include greedy searches, tabu searches, sim-

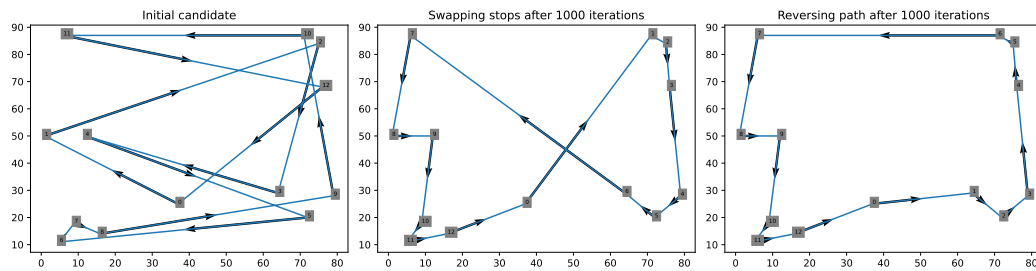


Figure 9.5 The final tours obtained by using the neighbourhood search in R

ulated annealing, genetic algorithms, and ant colony optimisation. They are usually employed when the problem is too large or complex to use exact methodologies.

The Travelling Salesman Problem, described in this chapter, is a classic example of one of these problems, formally described first in ??, although thought to have been discussed informally centuries before. It is an example of a large number of types of problems collectively known as vehicle routing problems, which often require heuristic methods for their solutions. A survey is given in (Kris Braekers, Katrien Ramaekers, and Inneke Van Nieuwenhuyse. “The vehicle routing problem: State of the art classification and review”. In: *Computers & Industrial Engineering* 99 [2016], pp. 300–313). Variations of the problem include multiple, heterogeneous and/or capacitated vehicles, and stochastic or time-dependent travel times. A recent adaptation of the problem is the green vehicle routing problem (Reza Moghdani et al. “The green vehicle routing problem: A systematic literature review”. In: *Journal of Cleaner Production* 279 [2021], p. 123691), where the cost function includes consideration of green house gas emissions and other pollutants.

For more diverse applications of heuristic methods, consider (Rhyd Lewis and Fiona Carroll. “Creating seating plans: a practical application”. In: *Journal of the Operational Research Society* 67.11 [2016], pp. 1353–1362) which describes a tabu search algorithm for finding seating plans for a wedding; and (Ruiju Tong et al. “Modeling the habitat suitability for deep-water gorgonian corals based on terrain variables”. In: *Ecological Informatics* 13 [2013], pp. 123–132) where a genetic algorithm is used to build a prediction model for locations of deep-sea wildlife habitats.





---

# Bibliography

---

- Axelrod, Robert and William Donald Hamilton. “The evolution of cooperation”. In: *science* 211.4489 (1981), pp. 1390–1396.
- Bozorg-Haddad, Omid, Mohammad Solgi, and Hugo A Loáiciga. *Meta-heuristic and evolutionary algorithms for engineering optimization*. John Wiley & Sons, 2017.
- Braekers, Kris, Katrien Ramaekers, and Inneke Van Nieuwenhuyse. “The vehicle routing problem: State of the art classification and review”. In: *Computers & Industrial Engineering* 99 (2016), pp. 300–313.
- Brailsford, Sally C et al. “An analysis of the academic literature on simulation and modelling in health care”. In: *Journal of simulation* 3.3 (2009), pp. 130–140.
- Brailsford, Sally C. et al. “Hybrid simulation modelling in operational research: A state-of-the-art review”. In: *European Journal of Operational Research* 278.3 (2019), pp. 721–737. ISSN: 0377-2217. DOI: <https://doi.org/10.1016/j.ejor.2018.10.025>. URL: <https://www.sciencedirect.com/science/article/pii/S0377221718308786>.
- Burden, Richard L, J Douglas Faires, and Albert C Reynolds. *Numerical analysis*. Brooks/cole Pacific Grove, CA, 2001.
- Conforti, Michele, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*. Vol. 271. Springer, 2014.
- Cooper, Ian, Argha Mondal, and Chris G Antonopoulos. “A SIR model assumption for the spread of COVID-19 in different communities”. In: *Chaos, Solitons & Fractals* 139 (2020), p. 110057.
- Coyle, JM, D Exelby, and JSystemDynamicsInDefenceAnalysis Holt. “System dynamics in defence analysis: some case studies”. In: *Journal of the Operational Research Society* 50.4 (1999), pp. 372–382.
- Croes, Georges A. “A method for solving traveling-salesman problems”. In: *Operations research* 6.6 (1958), pp. 791–812.
- Deo, Sarang and Itai Gurvich. “Centralized vs. decentralized ambulance diversion: A network perspective”. In: *Management Science* 57.7 (2011), pp. 1300–1319.
- Daz, Diego A, Ana Mara Jiménez, and Cristián Larroulet. “An agent-based model of school choice with information asymmetries”. In: *Journal of Simulation* 15.1-2 (2021), pp. 130–147.
- Daz-Parra, Ocotlán et al. “A survey of transportation problems”. In: *Journal of Applied Mathematics* 2014 (2014).
- Durán, Guillermo et al. “Scheduling the Chilean soccer league by integer programming”. In: *Interfaces* 37.6 (2007), pp. 539–552.

- Figueira, Gonçalo and Bernardo Almada-Lobo. “Hybrid simulation–optimization methods: A taxonomy and discussion”. In: *Simulation Modelling Practice and Theory* 46 (2014), pp. 118–134.
- Flynn, Michael J. “Very high-speed computing systems”. In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.
- Forrester, Jay W. “Industrial dynamics. 1961”. In: *Pegasus Communications, Waltham, MA* (1961).
- “The beginning of system dynamics”. In: (1995).
- Fudenberg, Drew et al. *The theory of learning in games*. Vol. 2. MIT press, 1998.
- Gardener, Martin. “MATHEMATICAL GAMES: The fantastic combinations of John Conway’s new solitaire game life”. In: *Scientific American* 223.4 (1970), pp. 120–123.
- Griffiths, Jeff D, Janet E Williams, and Richard Max Wood. “Modelling activities at a neurological rehabilitation unit”. In: *European Journal of Operational Research* 226.2 (2013), pp. 301–312.
- Guerriero, Francesca and Rosita Guido. “Operational research in the management of the operating theatre: a survey”. In: *Health care management science* 14.1 (2011), pp. 89–114.
- Hifi, Mhand et al. “A linear programming approach for the three-dimensional bin-packing problem”. In: *Electronic Notes in Discrete Mathematics* 36 (2010), pp. 993–1000.
- Jing, Peng et al. “Agent-based simulation of autonomous vehicles: A systematic literature review”. In: *IEEE Access* 8 (2020), pp. 79089–79103.
- Kerckhoffs, Auguste. “La cryptographie militaire. Journal des sciences militaires”. In: *IX (38)* 5 (1883).
- Lanchester, Frederick William. *Aircraft in warfare: The dawn of the fourth arm*. Constable limited, 1916.
- Lewis, Rhyd and Fiona Carroll. “Creating seating plans: a practical application”. In: *Journal of the Operational Research Society* 67.11 (2016), pp. 1353–1362.
- Martn Garca, J. *Theory and practical exercises of System Dynamics*. 2006.
- Maschler, Michael, Eilon Solan, and Shmuel Zamir. *Game theory*. Vol. 979. 2013, p. 4.
- Matsumoto, Makoto and Takuji Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”. In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8.1 (1998), pp. 3–30.
- Michalewicz, Zbigniew and David B Fogel. *How to solve it: modern heuristics*. Springer Science & Business Media, 2013.
- Moghdani, Reza et al. “The green vehicle routing problem: A systematic literature review”. In: *Journal of Cleaner Production* 279 (2021), p. 123691.
- Moler, Cleve and Charles Van Loan. “Nineteen dubious ways to compute the exponential of a matrix”. In: *SIAM review* 20.4 (1978), pp. 801–836.
- “Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later”. In: *SIAM review* 45.1 (2003), pp. 3–49.
- Nasar, Sylvia. *A beautiful mind*. Simon and Schuster, 2011.
- Nash, John. “Non-cooperative games”. In: *Annals of mathematics* (1951), pp. 286–295.

- Nash, John F et al. "Equilibrium points in n-person games". In: *Proceedings of the national academy of sciences* 36.1 (1950), pp. 48–49.
- Osorio, Andres F et al. "Simulation-optimization model for production planning in the blood supply chain". In: *Health care management science* 20.4 (2017), pp. 548–564.
- Press, William H and Freeman J Dyson. "Iterated Prisoners Dilemma contains strategies that dominate any evolutionary opponent". In: *Proceedings of the National Academy of Sciences* 109.26 (2012), pp. 10409–10413.
- Reynolds, Craig W. "Flocks, herds and schools: A distributed behavioral model". In: *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. 1987, pp. 25–34.
- Robinson, Clark. *Dynamical systems: stability, symbolic dynamics, and chaos*. CRC press, 1998.
- Robinson, Stewart. *Simulation: the practice of model development and use*. Vol. 50. Wiley Chichester, 2004.
- Romanowska, Iza et al. "Agent-based modeling for archaeologists: Part 1 of 3". In: *Advances in archaeological practice* 7.2 (2019), pp. 178–184.
- Särkkä, Simo and Arno Solin. *Applied stochastic differential equations*. Vol. 10. Cambridge University Press, 2019.
- Schelling, Thomas C. *Micromotives and macrobehavior*. WW Norton & Company, 2006.
- Schreuder, JAM. "Application of a location model to fire stations in Rotterdam". In: *European Journal of Operational Research* 6.2 (1981), pp. 212–219.
- Shoham, Yoav and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2008.
- Smith, J Maynard. "The theory of games and the evolution of animal conflicts". In: *Journal of theoretical biology* 47.1 (1974), pp. 209–221.
- Stewart, Ian. "Monopoly revisited". In: *Scientific American* 275.4 (1996), pp. 116–119.
- Stewart, James. *Calculus: Concepts and contexts*. Cengage Learning, 2009.
- Stewart, William J. *Probability, Markov chains, queues, and simulation*. Princeton university press, 2009.
- Sultan, Alan. *Linear programming: An introduction with applications*. Elsevier, 2014.
- Syms, Richard and Laszlo Solymar. "A dynamic competition model of regime change". In: *Journal of the Operational Research Society* 66.11 (2015), pp. 1939–1947.
- Tan, Bari. "Markov chains and the RISK board game". In: *Mathematics Magazine* 70.5 (1997), pp. 349–357.
- Tong, Ruiju et al. "Modeling the habitat suitability for deep-water gorgonian corals based on terrain variables". In: *Ecological Informatics* 13 (2013), pp. 123–132.
- Van Loan, Charles F and G Golub. *Matrix computations (Johns Hopkins studies in mathematical sciences)*. The Johns Hopkins University Press, 1996.
- Vandergraft, James S. "A fluid flow model of networks of queues". In: *Management Science* 29.10 (1983), pp. 1198–1208.

- Vázquez-Serrano, Jesús Isaac and RE Peimbert-García. “System dynamics applications in healthcare: A literature review”. In: *Proceedings of the international conference on industrial engineering and operations management*. 2020, pp. 10–12.
- Von Neumann, John. “13. various techniques used in connection with random digits”. In: *Appl. Math Ser* 12.36-38 (1951), p. 3.
- White, Douglas J. “A survey of applications of Markov decision processes”. In: *Journal of the operational research society* 44.11 (1993), pp. 1073–1096.
- Wilson, Greg et al. “Best practices for scientific computing”. In: *PLoS biology* 12.1 (2014), e1001745.