

---

*Half Title Page*



---

## *Title Page*

---

*LOC Page*

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



---

# Contents

---

Foreword	xi
Preface	xiii
Contributors	xv
SECTION I Getting Started	
CHAPTER 1 ■ Introduction	3
1.1 WHO IS THIS BOOK FOR?	3
1.2 WHAT DO WE MEAN BY APPLIED MATHEMATICS?	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	13
2.4 SOLVING WITH R	20
2.5 RESEARCH	27
CHAPTER 3 ■ Discrete Event Simulation	29
3.1 TYPICAL PROBLEM	29
3.2 THEORY	30
3.2.1 Event Scheduling Approach	32
3.2.2 Process Based Simulation	32

3.3	SOLVING WITH PYTHON	32
3.4	SOLVING WITH R	38
3.5	RESEARCH HIGHLIGHTS	44

### 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	RESEARCH	57

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

---

5.1	PROBLEM	59
5.2	THEORY	59
5.3	SOLVING WITH PYTHON	62
5.4	SOLVING WITH R	69
5.5	RESEARCH	76

### SECTION IV Emergent Behaviour

CHAPTER	6 ■ Game Theory	79
---------	-----------------	----

---

6.1	PROBLEM	79
6.2	THEORY	80
6.3	SOLVING WITH PYTHON	82
6.4	SOLVING WITH R	85
6.5	RESEARCH	88

CHAPTER	7 ■ Agent Based Simulation	89
---------	----------------------------	----

---

7.1	PROBLEM	89
7.2	THEORY	89
7.3	SOLVING WITH PYTHON	92
7.4	SOLVING WITH R	97
7.5	RESEARCH	103



**SECTION V Optimisation**

<b>CHAPTER</b>	<b>8 ■ Linear programming</b>	<b>107</b>
8.1	PROBLEM	107
8.2	THEORY	108
8.3	SOLVING WITH PYTHON	112
8.4	SOLVING WITH R	117
8.5	RESEARCH	127
<b>CHAPTER</b>	<b>9 ■ Heuristics</b>	<b>129</b>
9.1	PROBLEM	129
9.2	THEORY	129
9.3	SOLVING WITH PYTHON	132
9.4	SOLVING WITH R	140
9.5	RESEARCH	148
	<b>Bibliography</b>	<b>151</b>



---

# Foreword

---

This is the foreword



---

# Preface

---

This is the preface.



---

# Contributors

---

**Michaél Aftosmis**

NASA Ames Research Center  
Moffett Field, California

**Pratul K. Agarwal**

Oak Ridge National Laboratory  
Oak Ridge, Tennessee

**Sadaf R. Alam**

Oak Ridge National Laboratory  
Oak Ridge, Tennessee

**Gabrielle Allen**

Louisiana State University  
Baton Rouge, Louisiana

**Martin Sandve Alnæs**

Simula Research Laboratory and University  
of Oslo, Norway  
Norway

**Steven F. Ashby**

Lawrence Livermore National Laboratory  
Livermore, California

**David A. Bader**

Georgia Institute of Technology  
Atlanta, Georgia

**Benjamin Bergen**

Los Alamos National Laboratory  
Los Alamos, New Mexico

**Jonathan W. Berry**

Sandia National Laboratories  
Albuquerque, New Mexico

**Martin Berzins**

University of Utah

Salt Lake City, Utah

**Abhinav Bhatele**

University of Illinois  
Urbana-Champaign, Illinois

**Christian Bischof**

RWTH Aachen University  
Germany

**Rupak Biswas**

NASA Ames Research Center  
Moffett Field, California

**Eric Bohm**

University of Illinois  
Urbana-Champaign, Illinois

**James Bordner**

University of California, San Diego  
San Diego, California

**Geörge Bosilca**

University of Tennessee  
Knoxville, Tennessee

**Grèg L. Bryan**

Columbia University  
New York, New York

**Marian Bubak**

AGH University of Science and Technology  
Kraków, Poland

**Andrew Canning**

Lawrence Berkeley National Laboratory  
Berkeley, California

**xvi ■ Contributors**

**Jonathan Carter**

Lawrence Berkeley National Laboratory  
Berkeley, California

**Zizhong Chen**

Jacksonville State University  
Jacksonville, Alabama

**Joseph R. Crobak**

Rutgers, The State University of New  
Jersey

Piscataway, New Jersey

**Roxana E. Diaconescu**

Yahoo! Inc.  
Burbank, California

**Roxana E. Diaconescu**

Yahoo! Inc.  
Burbank, California



# I

---

## Getting Started



# Introduction

---

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

- Problems that can be solved using mathematics;
- Software that is free to use and change.

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

## 1.1 WHO IS THIS BOOK FOR?

---

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:

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. For example, one should never use any cryptographic software that is not open source: if you cannot open up and read things then you should not trust it (this is indeed why most cryptographic systems used are open source).

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

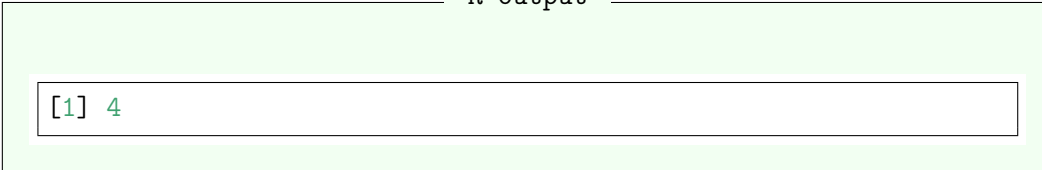
```
2 4
```

You will see R input as follows:

R input

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

and you will see R output as follows:

A screenshot of an R console output. It features a light green rectangular background. At the top center of this background, the text "R output" is written in a dark font. Below this text, there is a white rectangular box with a thin black border. Inside this white box, the text "[1] 4" is displayed in a green monospace font. To the left of the white box, the number "4" is printed in a small, dark font, indicating the line number of the output.

```
R output
```

```
[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. Interested readers might want to see Figure 1.1 which shows how the 3 principles interact with each other.

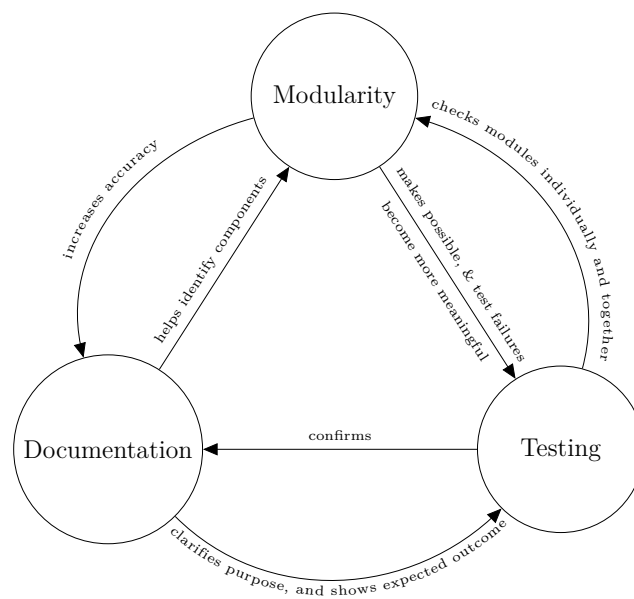


Figure 1.1 The relationship between modularisation, documentation and testing



# II

---

## Probabilistic Modelling



# Markov Chains

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

## 2.1 PROBLEM

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

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

This is represented diagrammatically in Figure 2.1.

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

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

## 2.2 THEORY

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

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

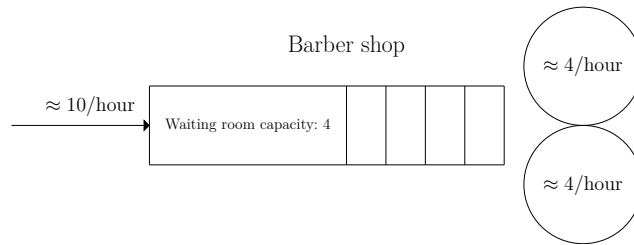


Figure 2.1 Diagrammatic representation of the barber shop as a queuing system.

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

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

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

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

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

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

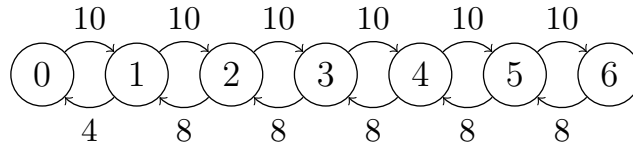


Figure 2.2 Diagrammatic representation of the state space and the transition rates

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

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

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

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

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

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

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

This long run probability distribution over the state can be represented using a vector  $\pi$  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.

## 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     capacity = waiting_room + num_barbers
25     delta = out_state - in_state
26
27     if delta == 1 and in_state < capacity:
28         return arrival_rate
29
30     if delta == -1:
31         return min(in_state, num_barbers) * service_rate
32
33     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

```

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

```

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

## Python input

```

69 Q = get_transition_rate_matrix()
70 print(Q)

```

which gives:

## Python output

```

71 [[-10  10  0  0  0  0  0]
72  [  4 -14 10  0  0  0  0]
73  [  0  8 -18 10  0  0  0]
74  [  0  0  8 -18 10  0  0]
75  [  0  0  0  8 -18 10  0]
76  [  0  0  0  0  8 -18 10]
77  [  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

```

78 import scipy.linalg
79
80 print(scipy.linalg.expm(Q * 0.5).round(5))

```

which gives:

## Python output

```

81 [[0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.0815 ]
82  [0.08501 0.18292 0.18666 0.1708  0.14377 0.1189  0.11194]
83  [0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346]
84  [0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142]
85  [0.02667 0.07361 0.10005 0.13422 0.17393 0.2189  0.27262]
86  [0.01567 0.0487  0.07552 0.11775 0.17512 0.24484 0.32239]
87  [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

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

which gives:

## Python output

```
89 [[0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
90  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
91  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
92  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
93  [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
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]]
```

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

```

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

```

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

## Python input

```

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

```

giving:

## Python output

```

112 [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

```

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

```

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

## Python input

```

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

```

which gives:

## Python output

```

132 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

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

which gives:

Python output

```
134 0.23557
```

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

Python input

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

Python output

```
136 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

```

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

```

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

## R input

```

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

```

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

## R input

```

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

```

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

## R input

```

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

```

which gives:

## R output

```

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

```

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

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

## R input

```

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

```

which gives:

## R output

```

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

```

After 500 time units:

## R input

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

which gives:

## R output

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

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

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

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



## R input

```

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

```

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

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

## R input

```

234 print(get_steady_state_vector(Q))

```

giving:

## R output

```

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

```

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

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

R input

```

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

```

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

R input

```

262 print(get_probability_of_full_shop())

```

which gives:

R output

```

263 [1] 0.261756

```

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

Adding 2 extra spaces in the waiting rooms corresponds to:

R input

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

which decreases the probability of a full shop to:

R output

```
265 [1] 0.2355699
```

but adding another barber and chair:

R input

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

gives:

R output

```
267 [1] 0.0786359
```

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

## 2.5 RESEARCH

---

TBA



# Discrete Event Simulation

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

## 3.1 TYPICAL PROBLEM

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

- Bicycles arrive randomly at the shop at a rate of 15 per hour.
- They wait in line to be seen at an inspection counter, staffed by one member of staff who can inspect one bicycle at a time. On average an inspection takes around 3 minutes.
- Around 20% of bicycles do not need repair after inspection, and they are then ready for collection.
- Around 80% of bicycles go on to be repaired after inspection. These then wait in line outside the repair workshop, which is staffed by two members of staff who can each repair one bicycle at a time. On average a repair takes around 6 minutes.
- After repair the bicycles are ready for collection.

A diagram of the system is shown in Figure 3.1.

An assumption of infinite capacity at the bicycle repair shop for waiting bicycles is made. The shop will hire an extra member of staff in order to meet their target of a maximum time in the system of 30 minutes. They would like to know if they should work on the inspection counter or in the repair workshop?

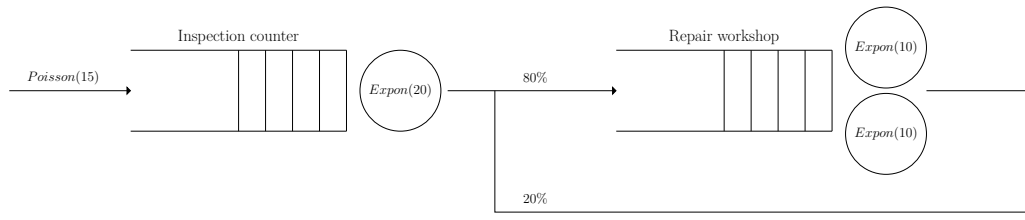


Figure 3.1 Diagrammatic representation of the bicycle repair shop as a queuing system.

### 3.2 THEORY

A number of aspects of the bicycle shop above are probabilistic. For example the times that bicycles arrive at the shop, the duration of the inspection and repairs, and whether the bicycle would need to go on to be repaired or not. When a number of these probabilistic events are linked together such as the bicycle shop a method to model this situation is *Discrete Event Simulation*.

Consider one probabilistic event, rolling a six sided die where each side is equally likely to land. Therefore the probability of rolling a 1 is  $\frac{1}{6}$ , the probability of rolling a 2 is  $\frac{1}{6}$ , and so on. This means that that if the die is rolled a large number of times,  $\frac{1}{6}$  of those rolls would be expected to be a 1.

Consider a random process in which the actual values of the probability of events occurring are not known. Consider rolling a weighted die, in this case a die in which the probability of obtaining one number is much greater than the others. How can probability of obtaining a 1 on this die be estimated?

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

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

In order to do this, the computer needs to be able to generate random outcomes of each of the smaller events that make up the large complex system. Generating

random events are essentially doing things with random numbers, these need to be generated.

Computers are deterministic, therefore true randomness is in itself a challenging mathematical problem. They can however generate pseudorandom numbers: sequences of numbers that look like random numbers, but are entirely determined from the previous numbers in the sequence. Most programming languages have methods of doing this.

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

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

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

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

- the time each bicycle arrives to the repair shop,
- the time each bicycle spends at the inspection counter,
- whether each bicycle needs to go on to the repair workshop,
- the time those bicycles spend being repaired.

As the simulation progresses these events will be generated, and will interact together as described in Section 9.1. The proportion of customers spending longer than 30 minutes in the shop can then be counted. This proportion itself is a random variable, and so like the weighted die, running this simulation once does not give much information. The simulation can be run many times and to give an average proportion.

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

### 3.2.1 Event Scheduling Approach

When using the event scheduling approach, the ‘virtual representation’ of the system is the collection of facilities that the bicycles use, shown in Figure 3.1. Then the entities (the bicycles) interact with these facilities. It is these facilities that determine how the entities behave.

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

### 3.2.2 Process Based Simulation

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

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

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

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

## 3.3 SOLVING WITH PYTHON

In this book the Ciw library will be used in order to conduct Discrete Event Simulation in Python. Ciw uses the event scheduling approach, which means the system’s facilities are defined, and customers then interact with them.

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

- the distribution of times between consecutive bicycles arriving,
- the distribution of times the bicycles spend in service,
- the number of servers available,
- the probability of routing to each of the other facilities after service.

In this case the time between consecutive arrivals will be assumed to follow an



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

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

#### Python input

```

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

```

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

## Python input

```

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

```

which gives:

## Python output

```

303 2

```

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

## Python input

```

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

```

Notice here a random seed is set. This is because there is randomness in running the simulation, setting a seed ensures reproducible results. Notice also that the simulation always begins with an empty system, so the first bicycle to arrive will never wait for service. Depending on the situation this may be an unwanted feature, though not in this case as it is reasonable to assume that the bicycle shop will begin the day with no customers.

To count the number of bicycles that have finished service, and to count the number of those whose entire journey through the system lasted longer than 0.5 hours the `pandas` library will be used:

## Python input

```

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

```

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

## Python input

```

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

```

This gives:

## Python output

347

0.261261

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

However this particular day may have contained a number of extreme events. For a more accurate proportion this experiment should be repeated a number of times, and an average proportion taken. The following function returns an average proportion:

## Python input

348

```
def get_average_proportion(num_inspectors=1, num_repairers=2):
    """Returns the average proportion of bicycles spending over
    a given limit at the repair shop.

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

    Returns:
        a real
    """
    num_trials = 100
    N = build_network_object(
        num_inspectors=num_inspectors,
        num_repairers=num_repairers,
    )
    proportions = []
    for trial in range(num_trials):
        Q = run_simulation(N, seed=trial)
        proportion = get_proportion(Q=Q)
        proportions.append(proportion)
    return sum(proportions) / num_trials
```

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

## Python input

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

which gives:

## Python output

```
372 0.159354
```

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

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

## Python input

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

which gives:

## Python output

```
375 0.038477
```

that is 3.85% of bicycles.

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

Python input

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

which gives:

Python output

```
378 0.103591
```

that is 10.36% of bicycles.

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

### 3.4 SOLVING WITH R

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

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

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

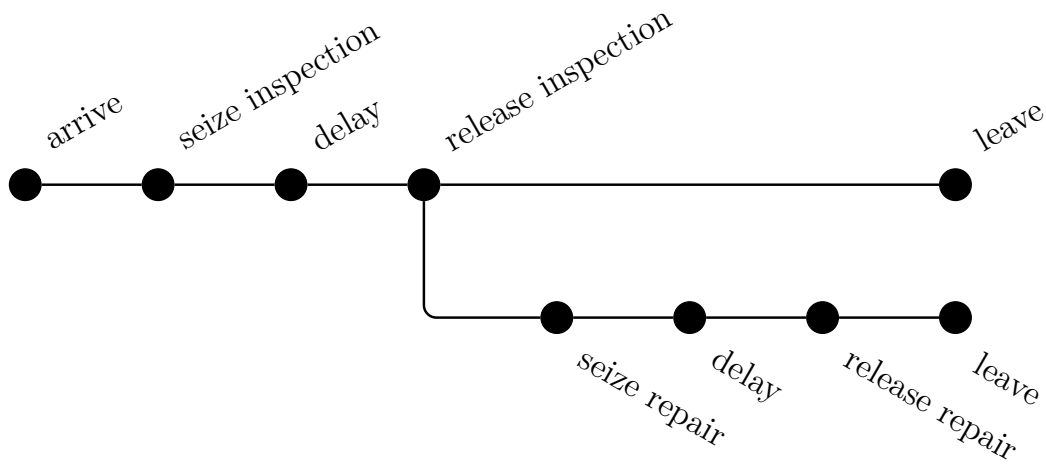


Figure 3.2 Diagrammatic representation of the forked trajectories a bicycle can take

#### R input

```

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

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

```

405 #' Runs one trial of the simulation.
406 #'
407 #' @param bicycle a simmer trajectory object
408 #' @param num_inspectors positive integer (default: 1)
409 #' @param num_repairers positive integer (default: 2)
410 #' @param seed a float (default: 0)
411 #'
412 #' @return A simmer simulation object after one run of
413 #' the simulation
414 run_simulation <- function(bicycle,
415                             num_inspectors = 1,
416                             num_repairers = 2,
417                             seed = 0) {
418   arrival_rate <- 15
419   max_time <- 8
420   repair_shop <-
421     simmer("Repair Shop") %>%
422     add_resource("Inspector", num_inspectors) %>%
423     add_resource("Repairer", num_repairers) %>%
424     add_generator(
425       "Bicycle", bicycle, function() {
426         rexp(1, arrival_rate)
427       }
428     )
429   set.seed(seed)
430   repair_shop %>% run(until = 8)
431   return(repair_shop)
432 }

```

Notice here a random seed is set. This is because there are elements of randomness when running the simulation, setting a seed ensures reproducible results. Notice also that the simulation always begins with an empty system, so the first bicycle to arrive will never wait for service. Depending on the situation this may be an unwanted feature, though not in this case as it is reasonable to assume that the bicycle shop will begin the day with no customers.



To count the number of bicycles that have finished service, and to count the number of those whose entire journey through the system lasted longer than 0.5 hours, Simmer's `get_mon_arrivals()` function gives a data frame that can be manipulated:

R input

```

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

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

```

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

This piece of code gives

R output

```

448 [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

```

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

```

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

R input

```

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

```

which gives:

## R output

```
480 [1] 0.1635779
```

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

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

## R input

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

which gives:

## R output

```
487 [1] 0.04221602
```

that is 4.22% of bicycles.

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

## R input

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

which gives:

R output

```
494 [1] 0.1224761
```

that is 12.25% of bicycles.

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

### 3.5 RESEARCH HIGHLIGHTS

---

# III

## Dynamical Systems



# 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 define the symbolic variables that will be used. The Python library SymPy is used which allows symbolic calculations.

Python input

```

495 import sympy as sym
496
497 t = sym.Symbol("t")
498 alpha = sym.Symbol("alpha")
499 I_0 = sym.Symbol("I_0")
500 I = sym.Function("I")

```

Now write a function to obtain the differential equation.



## Python input

```

501 def get_equation(alpha=alpha):
502     """Return the differential equation.
503
504     Args:
505         alpha: a float (default: symbolic alpha)
506
507     Returns:
508         A symbolic equation
509     """
510     return sym.Eq(sym.Derivative(I(t), t), -alpha * I(t))

```

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

## Python input

```

511 eq = get_equation()
512 print(eq)

```

which gives:

## Python output

```

513 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

```

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

```

Python output

```
516 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

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

This can verify the solution discussed previously:

Python input

```
529 sol = get_solution()
530 print(sol)
```

which gives:

Python output

```
531 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

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

which gives:

## Python output

```
533 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

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

which gives:

## Python output

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

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

## Python input

```

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

```

The cost without purchasing the cure is:

## Python input

```

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

```

which gives:

## Python output

```

564 500

```

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

## Python input

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

which gives:

## Python output

```
570 750
```

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

```

571 #' Returns the numerical value of the derivative.
572 #'
573 #' @param t a set of time points
574 #' @param y a function
575 #' @param parameters the set of all parameters passed to y
576
577 #' @return a float
578 derivative <- function(t, y, parameters) {
579   with(
580     as.list(c(y, parameters)), {
581       dIdt <- -alpha * I # nolint
582       list(dIdt) # nolint
583     }
584   )
585 }

```

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

## R input

```

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

```

This gives:

## R output

```

587 [[1]]
588 [1] -200

```

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

## R input

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

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

```

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

```

The cost without purchasing the cure is:

## R input

```

634 alpha <- 2
635 cost_without_cure <- get_cost(alpha = alpha)
636 print(round(cost_without_cure))

```



which gives:

R output

```
637 [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

```
638 cost_of_cure <- 5
639 cost_with_cure <- get_cost(
640   alpha = 2 * alpha, cost_of_cure = cost_of_cure
641 )
642 print(round(cost_with_cure))
```

which gives:

R output

```
643 [1] 750
```

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

## 4.5 RESEARCH

TBA



# Systems Dynamics

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

## 5.1 PROBLEM

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

- They have a birth rate  $b$  of 0.01 per day;
- They have a death rate  $d$  of 0.01 per day;
- For every infectious individual, the infection rate  $\alpha$  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.

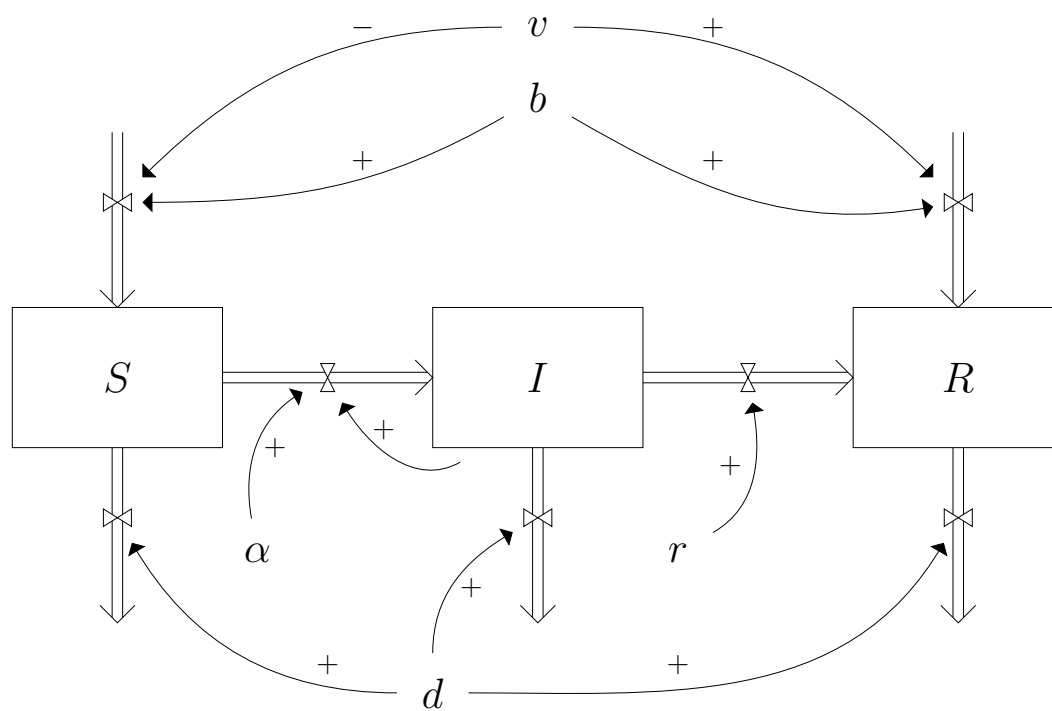


Figure 5.1 Diagrammatic representation of the epidemiology model

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, 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.

## Python input

```

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

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

## Python output

```
679 (-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

```

680 from scipy.integrate import odeint
681
682
683 def integrate_ode(
684     derivative_function,
685     t,
686     y0=(2999, 1, 0),
687     vaccine_rate=0.85,
688     birth_rate=0.01,
689 ):
690     """Numerically solve the system of differential equations.
691
692     Args:
693         derivative_function: a function returning a tuple
694             of three floats
695         t: an array of increasing positive floats
696         y0: a tuple of three integers (default: (2999, 1, 0))
697         vaccine_rate: a positive float <= 1 (default: 0.85)
698         birth_rate: a positive float <= 1 (default: 0.01)
699
700     Returns:
701         A tuple of three arrays
702     """
703     results = odeint(
704         derivative_function,
705         y0,
706         t,
707         args=(vaccine_rate, birth_rate),
708     )
709     S, I, R = results.T
710     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%:

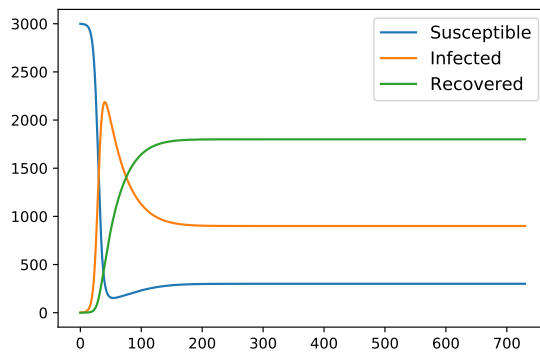


Figure 5.2 Output of code line 737-742

## Python input

```

711 import numpy as np
712 from scipy.integrate import odeint
713
714 t = np.arange(0, 730.01, 0.01)
715 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

```

716 import matplotlib.pyplot as plt
717
718 fig, ax = plt.subplots(1)
719 ax.plot(t, S, label='Susceptible')
720 ax.plot(t, I, label='Infected')
721 ax.plot(t, R, label='Recovered')
722 ax.legend(fontsize=12)
723 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

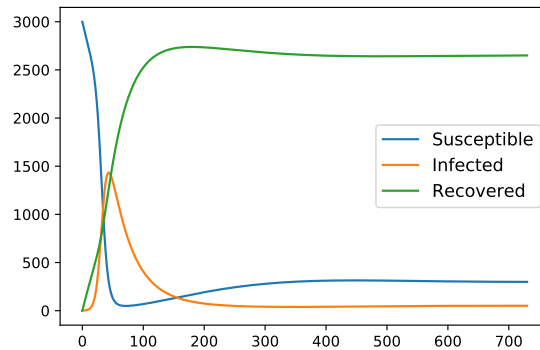


Figure 5.3 Output of code line 745-750

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 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

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

The following code gives the plot shown in Figure 5.3.

#### Python input

```
726 fig, ax = plt.subplots(1)
727 ax.plot(t, S, label='Susceptible')
728 ax.plot(t, I, label='Infected')
729 ax.plot(t, R, label='Recovered')
730 ax.legend(fontsize=12)
731 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

```

732 def daily_cost(
733     derivative_function=derivatives, vaccine_rate=0.85
734 ):
735     """Calculates the daily cost to the public health system
736     after 2 years.
737
738     Args:
739         derivative_function: a function returning a tuple
740             of three floats
741         vaccine_rate: a positive float <= 1 (default: 0.85)
742
743     Returns:
744         the daily cost
745     """
746     max_time = 730
747     time_step = 0.01
748     birth_rate = 0.01
749     vaccine_cost = 220
750     medication_cost = 10
751     t = np.arange(0, max_time + time_step, time_step)
752     S, I, R = integrate_ode(
753         derivatives,
754         t,
755         vaccine_rate=vaccine_rate,
756         birth_rate=birth_rate,
757     )
758     N = S[-1] + I[-1] + R[-1]
759     daily_vaccine_cost = (
760         N * birth_rate * vaccine_rate * vaccine_cost
761     ) / time_step
762     daily_meds_cost = (I[-1] * medication_cost) / time_step
763     return daily_vaccine_cost + daily_meds_cost

```

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

## Python input

```
764 cost = daily_cost(vaccine_rate=0.0)
765 print(round(cost, 2))
```

which gives

## Python output

```
766 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

```
767 cost = daily_cost(vaccine_rate=0.85)
768 print(round(cost, 2))
```

which gives

## Python output

```
769 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

```

770 #' Defines the system of differential equations that describe
771 #' the epidemiology model.
772 #'
773 #' @param t a positive float
774 #' @param y a tuple of three integers
775 #' @param vaccine_rate a positive float <= 1
776 #' @param birth_rate a positive float <= 1
777 #'
778 #' @return a list containing dS, dI, and dR
779 derivatives <- function(t, y, parameters){
780   infection_rate <- 0.3
781   recovery_rate <- 0.02
782   death_rate <- 0.01
783   with(
784     as.list(c(y, parameters)), {
785       N <- S + I + R
786       dSdt <- ( - ( (infection_rate * S * I) / N) # nolint
787         + ( (1 - vaccine_rate) * birth_rate * N)
788         - (death_rate * S))
789       dIdt <- ( ( (infection_rate * S * I) / N) # nolint
790         - (recovery_rate * I)
791         - (death_rate * I))
792       dRdt <- ( (recovery_rate * I) # nolint
793         - (death_rate * R)
794         + (vaccine_rate * birth_rate * N))
795       list(c(dSdt, dIdt, dRdt)) # nolint
796     }
797   )
798 }

```

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

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

## R output

```
803 [[1]]  
804 [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

```

805 library(deSolve) # nolint
806
807 #' Numerically solve the system of differential equations
808 #'
809 #' @param t an array of increasing positive floats
810 #' @param y0 list of integers (default: c(S=2999, I=1, R=0))
811 #' @param birth_rate a positive float <= 1 (default: 0.01)
812 #' @param vaccine_rate a positive float <= 1 (default: 0.85)
813 #'
814 #' @return a matrix of times, S, I and R values
815 integrate_ode <- function(times,
816                             y0 = c(S = 2999, I = 1, R = 0),
817                             birth_rate = 0.01,
818                             vaccine_rate = 0.84){
819   params <- c(birth_rate = birth_rate,
820               vaccine_rate = vaccine_rate)
821   ode(y = y0,
822       times = times,
823       func = derivatives,
824       parms = params)
825 }

```

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%:

## R input

```

826 times <- seq(0, 730, by = 0.01)
827 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.



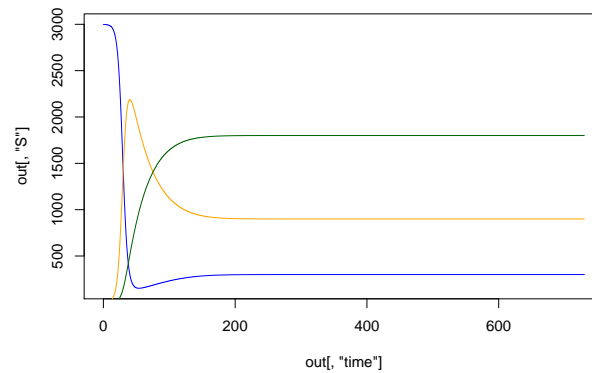


Figure 5.4 Output of code line 846-850

## R input

```

828 pdf("plot_no_vaccine_R.pdf", width = 7, height = 5)
829 plot(out[, "time"], out[, "S"], type = "l", col = "blue")
830 lines(out[, "time"], out[, "I"], type = "l", col = "orange")
831 lines(out[, "time"], out[, "R"], type = "l", col = "darkgreen")
832 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%:

## R input

```

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

```

The following code gives the plot shown in Figure 5.5.

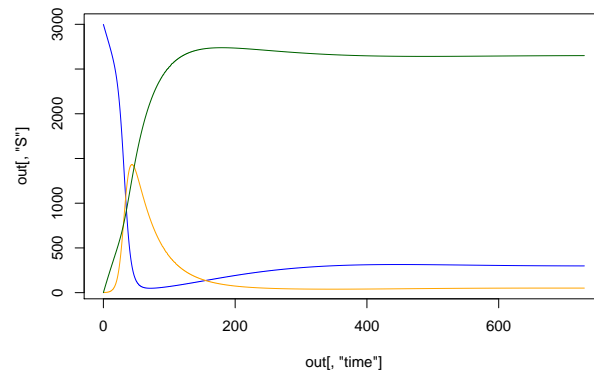


Figure 5.5 Output of code line 853-857

R input

```

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

```

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

```

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

## R input

```

864 cost <- daily_cost(vaccine_rate = 0.0)
865 print(cost)

```

which gives

## R output

```

866 [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

```
867 cost <- daily_cost(vaccine_rate = 0.85)
868 print(cost)
```

which gives

R output

```
869 [1] 611903.4
```

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

## 5.5 RESEARCH

---

# IV

---

## Emergent Behaviour



# Game Theory

MOST 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.

		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.

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 ??

		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.

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:

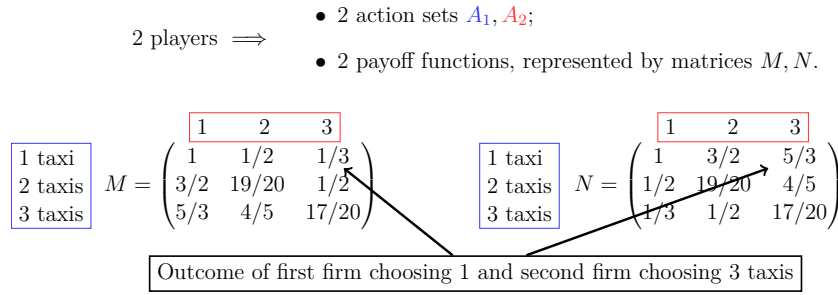
1. A given collection of actors that make decisions (players);
2. Options available to each player (actions);
3. A numerical value associated to each player for every possible choice of action made by all the players. This is the utility or reward.

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





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

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. 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, learning algorithms and/or agent based modelling.

### 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.

Python input

```

870 import nashpy as nash
871 import numpy as np
872
873
874 def get_game(profits, offset=0):
875     """Return the game object with a given offset when 3 taxis
876     are provided.
877
878     Args:
879         profits: a matrix with expected profits
880         offset: a float
881
882     Returns:
883         A nashpy game object
884     """
885     new_profits = np.array(profits)
886     new_profits[2] += offset
887     return nash.Game(new_profits, new_profits.T)

```

This gives the game for the considered problem:

Python input

```

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

```

which gives:

#### Python output

```

899 Bi matrix game with payoff matrices:
900
901 Row player:
902 [[1.          0.5          0.33333333]
903  [1.5         0.95         0.5         ]
904  [1.66666667 0.8          0.85         ]]
905
906 Column player:
907 [[1.          1.5          1.66666667]
908  [0.5         0.95         0.8         ]
909  [0.33333333 0.5          0.85         ]]

```

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

#### Python input

```

910 def get_equilibria(profits, offset=0):
911     """Return the equilibria for a given offset when 3 taxis
912     are provided.
913
914     Args:
915         profits: a matrix with expected profits
916         offset: a float
917
918     Returns:
919         A tuple of Nash equilibria
920     """
921     game = get_game(profits=profits, offset=offset)
922     return tuple(game.support_enumeration())

```

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

#### Python input

```

923 equilibria = get_equilibria(profits=profits)

```

The equilibria are:

## Python input

```

924 for eq in equilibria:
925     print(eq)

```

giving:

## Python output

```

926 (array([0., 1., 0.]), array([0., 1., 0.]))
927 (array([0., 0., 1.]), array([0., 0., 1.]))
928 (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

```

929 offset = 0
930 while len(get_equilibria(profits=profits, offset=offset)) > 1:
931     offset += 0.01

```

This gives a final offset value of:

## Python input

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

## Python output

```
933 0.15
```

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

## Python input

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

giving:

## Python output

```
935 ((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

```

936 library(Recon)
937
938 #' Returns the equilibria in pure strategies
939 #' for a given offset
940 #'
941 #' @param profits: a matrix with expected profits
942 #' @param offset: a float
943 #'
944 #' @return a list of equilibria
945 get_equilibria <- function(profits, offset = 0){
946   new_profits <- rbind(
947     profits[c(1, 2), ],
948     profits[3, ] + offset
949   )
950   sim_nasheq(new_profits, t(new_profits))
951 }

```

This gives the pure Nash equilibria:

## R input

```

952 profits <- rbind(
953   c(1, 1 / 2, 1 / 3),
954   c(3 / 2, 19 / 20, 1 / 2),
955   c(5 / 3, 4 / 5, 17 / 20)
956 )
957 eqs <- get_equilibria(profits = profits)
958 print(eqs)

```

which gives:

## R output

```

959 $`Equilibrium 1`
960 [1] "2" "2"
961
962 $`Equilibrium 2`
963 [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

```

964 offset <- 0
965 while (length(
966   get_equilibria(profits = profits, offset = offset)
967 ) > 1){
968   offset <- offset + 0.01
969 }

```

This gives a final offset value of:

R input

```

970 print(round(offset, 2))

```

R output

```

971 [1] 0.15

```

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

R input

972

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

giving:

R output

973

```
$`Equilibrium 1`
```

974

```
[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 RESEARCH

---

TBA



# 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. 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’, 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;
- An environment contains a number of agents and defines how their interactions affect each other. The agents may be homogeneous or heterogeneous, and the

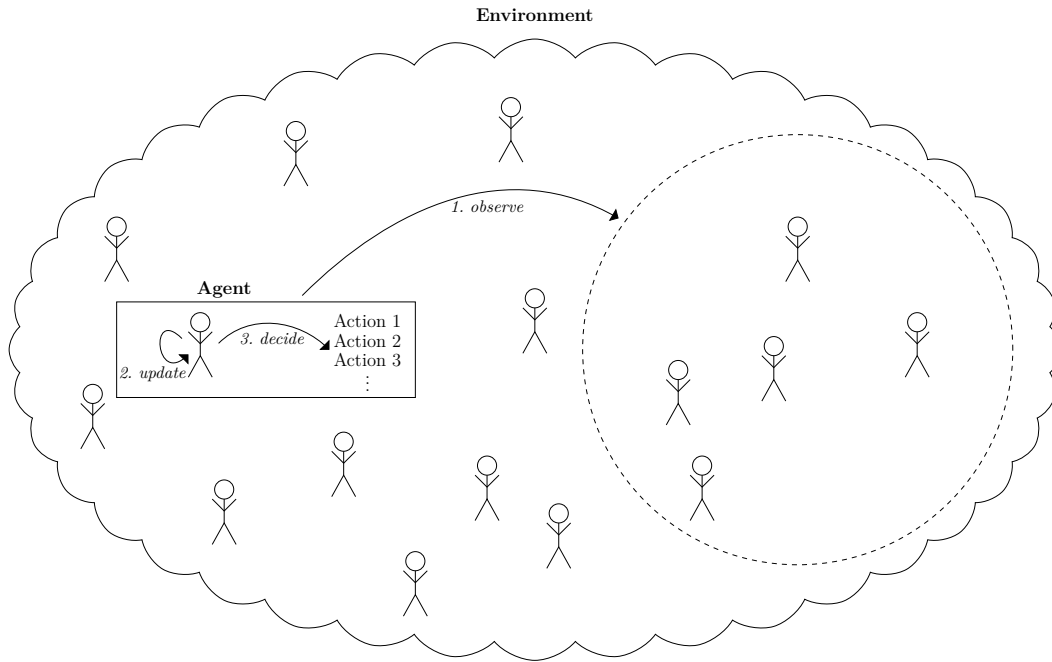


Figure 7.1 Representation of an agent interacting with its environment.

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 mathematical simplicity, it is also assumed that the grid is a torus, where houses in

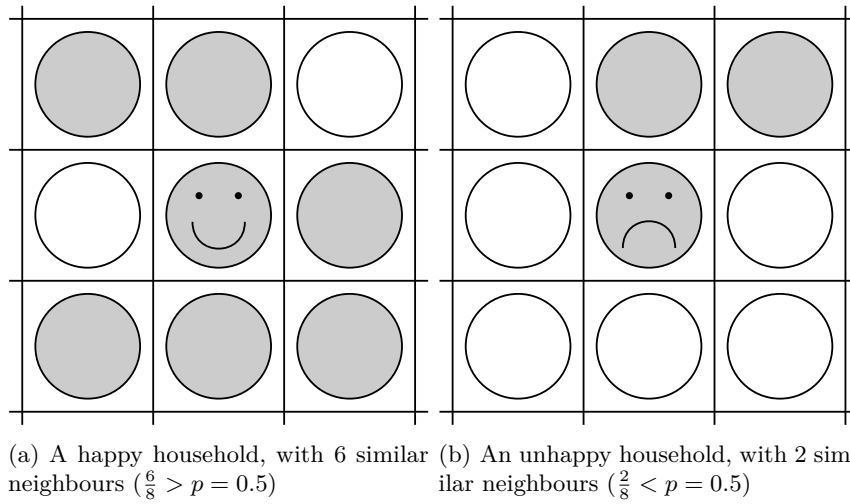


Figure 7.2 Example of a household happy and unhappy with its neighbours, when  $p = 0.5$ . Households supporting Cardiff City FC are shaded grey, households supporting Swansea City AFC are white.

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 with. 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

```
975 import random
976 import itertools
977 import numpy as np
```

Now to define the **City**:

## Python input

```

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

```

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

```

1017 class House:
1018     def __init__(self, x, y, threshold, city):
1019         """Initialises the House object.
1020             Args:
1021                 x: the integer x-coordinate
1022                 y: the integer y-coordinate
1023                 threshold: a number between 0 and 1 representing
1024                     the minimum acceptable proportion of similar
1025                     neighbours
1026                 city: an instance of the City class
1027             """
1028         self.x = x
1029         self.y = y
1030         self.threshold = threshold
1031         self.kind = random.choice(["Cardiff", "Swansea"])
1032         self.city = city
1033
1034     def satisfaction(self):
1035         """Determines the household's satisfaction level.
1036             Returns:
1037                 A proportion
1038             """
1039         same = 0
1040         for x, y in itertools.product([-1, 0, 1], [-1, 0, 1]):
1041             ax = (self.x + x) % self.city.size
1042             ay = (self.y + y) % self.city.size
1043             same += self.city.houses[ax, ay].kind == self.kind
1044         return (same - 1) / 8
1045
1046     def sad(self):
1047         """Determines if the household is sad.
1048             Returns:
1049                 a Boolean
1050             """
1051         return self.satisfaction() < self.threshold
1052
1053     def swap(self, house):
1054         """Swaps two households.
1055             Args:
1056                 house: the house object to swap household with
1057             """
1058         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

```

1059 def find_mean_happiness(seed, size, threshold, n_steps):
1060     """Create and run an instance of the simulation.
1061
1062     Args:
1063         seed: the random seed to use
1064         size: an integer number of rows and columns
1065         threshold: a number between 0 and 1 representing
1066             the minimum acceptable proportion of similar
1067             neighbours
1068         n_steps: an integer number of steps
1069
1070     Returns:
1071         The average city's household satisfaction after
1072         n_steps
1073     """
1074     random.seed(seed)
1075     C = City(size, threshold)
1076     C.run(n_steps)
1077     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

```
1078 print(find_mean_happiness(0, 50, 0.65, 0))
```

Python output

```
1079 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

```
1080 print(find_mean_happiness(0, 50, 0.65, 100))
```

Python output

```
1081 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, that overall emergent segregation levels are much higher than any individuals' personal preference for segregation.

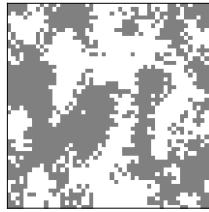
More analysis methods can be added, including plotting functions. Figure 7.3 shows the grid at the beginning, after 20 time steps, and after 100 time steps, with households supporting Cardiff City FC in grey, and those supporting Swansea City AFC in white. It visually shows the households segregating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

## 7.4 SOLVING WITH R

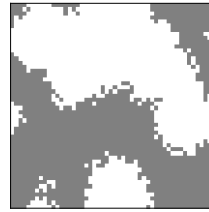
Agent based modelling lends itself well to a programming paradigm called object-orientated programming. This paradigm lets a number of *objects* from a set of instructions called a *class* to be built. These objects can both store information (in the R library used here these are called *fields*), and do things (in the R library used here



(a) At the beginning.



(b) After 20 time steps.



(c) After 100 time steps.

Figure 7.3 Plotted results from the Python code.

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**:

## R input

```

1082 library(R6)
1083 city <- R6Class("City", list(
1084   size = NA,
1085   houses = NA,
1086   initialize = function(size, threshold) {
1087     self$size <- size
1088     self$houses <- c()
1089     for (x in 1:size) {
1090       row <- c()
1091       for (y in 1:size) {
1092         row <- c(row, house$new(x, y, threshold, self))
1093       }
1094       self$houses <- rbind(self$houses, row)
1095     } },
1096   run = function(n_steps) {
1097     if (n_steps > 0) {
1098       for (turn in 1:n_steps) {
1099         self$take_turn()
1100       } },
1101   take_turn = function() {
1102     sad <- c()
1103     for (house in self$houses) {
1104       if (house$sad()) {
1105         sad <- c(sad, house)
1106       } }
1107     sad <- sample(sad)
1108     num_sad <- length(sad)
1109     i <- 1
1110     while (i <= num_sad / 2) {
1111       sad[[i]]$swap(sad[[num_sad - i]])
1112       i <- i + 1
1113     } },
1114   mean_satisfaction = function() {
1115     mean(sapply(self$houses, function(x) x$satisfaction()))
1116   })
1117 )

```

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

```

1118 house <- R6Class("House", list(
1119   x = NA,
1120   y = NA,
1121   threshold = NA,
1122   city = NA,
1123   kind = NA,
1124   initialize = function(x = NA,
1125                         y = NA,
1126                         threshold = NA,
1127                         city = NA) {
1128     self$x <- x
1129     self$y <- y
1130     self$threshold <- threshold
1131     self$city <- city
1132     self$kind <- sample(c("Cardiff", "Swansea"), 1)
1133   },
1134   satisfaction = function() {
1135     same <- 0
1136     for (x in -1:1) {
1137       for (y in -1:1) {
1138         ax <- ( (self$x + x - 1) %% self$city$size) + 1
1139         ay <- ( (self$y + y - 1) %% self$city$size) + 1
1140         if (self$city$houses[[ax, ay]]$kind == self$kind) {
1141           same <- same + 1
1142         } } }
1143     (same - 1) / 8
1144   },
1145   sad = function() {
1146     self$satisfaction() < self$threshold
1147   },
1148   swap = function(house) {
1149     old <- self$kind
1150     self$kind <- house$kind
1151     house$kind <- old
1152   })
1153 )

```

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

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

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

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

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

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

```

1154  #' Create and run an instance of the simulation.
1155  #'
1156  #' @param seed: the random seed to use
1157  #' @param size: an integer number of rows and columns
1158  #' @param threshold: a number between 0 and 1 representing
1159  #'   the minimum acceptable proportion of similar neighbours
1160  #' @param n_steps: an integer number of steps
1161  #'
1162  #' @return The average city's household satisfaction
1163  #'   after n_steps
1164  find_mean_happiness <- function(seed,
1165                                   size,
1166                                   threshold,
1167                                   n_steps){
1168    set.seed(seed)
1169    our_city <- city$new(size, threshold)
1170    our_city$run(n_steps)
1171    our_city$mean_satisfaction()
1172  }

```

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

```

1173  print(find_mean_happiness(0, 50, 0.65, 0))

```

R output

```
1174 [1] 0.4956
```

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

R input

```
1175 print(find_mean_happiness(0, 50, 0.65, 100))
```

R output

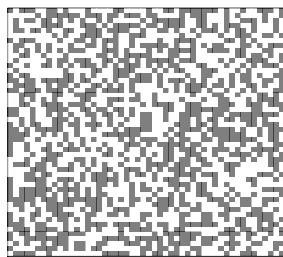
```
1176 [1] 0.9338
```

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

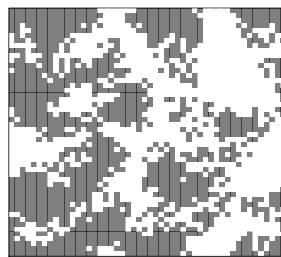
More analysis methods can be added, including plotting functions. Figure 7.4 shows the grid at the beginning, after 20 time steps, and after 100 time steps, with households supporting Cardiff City FC in grey, and those supporting Swansea City AFC in white. It shows the households segregating over time.

## 7.5 RESEARCH

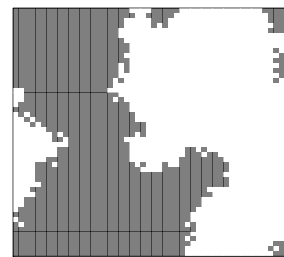
---



(a) At the beginning.



(b) After 20 time steps.



(c) After 100 time steps.

Figure 7.4 Plotted results from the R code.



# V

## Optimisation



# Linear Programming

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

## 8.1 PROBLEM

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

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

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

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

Table 8.1 List of modules on offer at the university.

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

How can every exam be scheduled with no clashes, 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. 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 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.

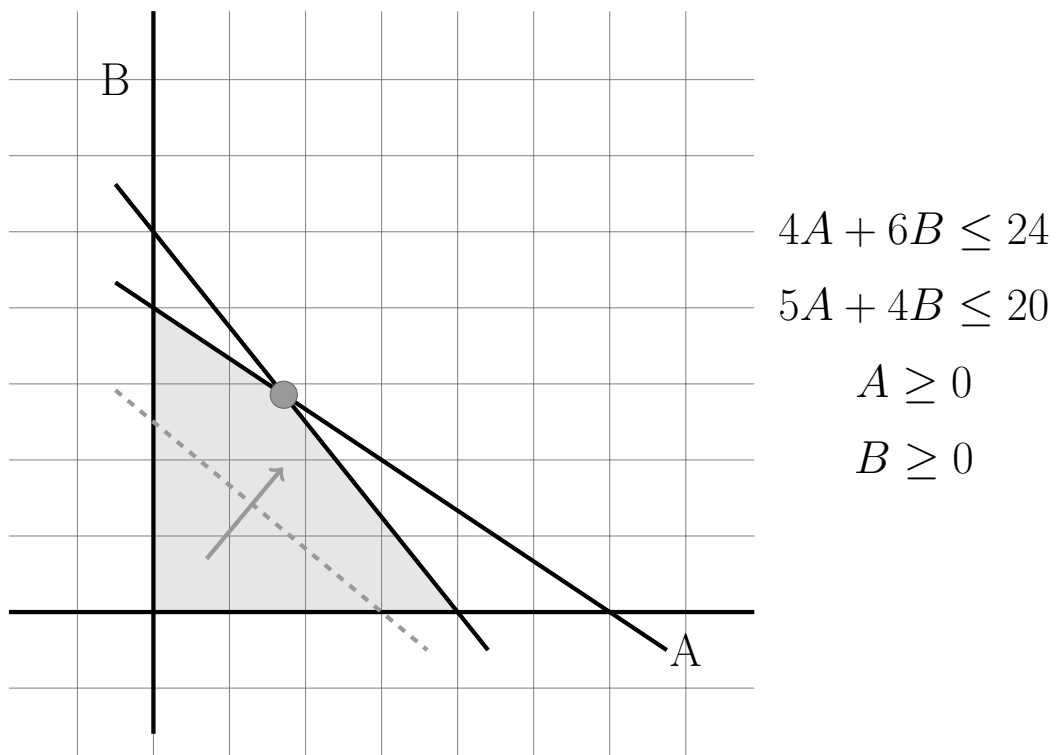


Figure 8.1 Visual representation of the paint linear program. The feasible convex region is shaded in grey; the objective function with arbitrary value is shown in a dashed line.

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.

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

```

1177 import pulp
1178
1179
1180 def get_variables(modules, times):
1181     """Returns the binary variables for a given timetabling
1182     problem.
1183
1184     Args:
1185         modules: The complete collection of modules to be
1186             timetabled.
1187         times: The collection of available time slots.
1188
1189     Returns:
1190         A tuple containing the decision variables x and y.
1191     """
1192     xshape = (modules, times)
1193     x = pulp.LpVariable.dicts("X", xshape, cat=pulp.LpBinary)
1194     y = pulp.LpVariable.dicts("Y", times, cat=pulp.LpBinary)
1195     return x, y

```

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

Python input

```

1196 Ac = [0, 1]
1197 Ao = [2, 3, 4]
1198 Bc = [5, 6]
1199 Bo = [7, 8]
1200 Cc = [9, 10]
1201 Co = [11, 12, 13]
1202 modules = Ac + Ao + Bc + Bo + Cc + Co
1203 times = range(14)
1204 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

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

Python output

```
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

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

Python output

```
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

```

1209 def get_solution(Ac, Ao, Bc, Bo, Cc, Co, times):
1210     """Returns the binary variables corresponding to the
1211     solution of given timetabling problem.
1212
1213     Args:
1214         Ac: The set of core art modules
1215         Ao: The set of optional art modules
1216         Bc: The set of core biology modules
1217         Bo: The set of optional biology modules
1218         Cc: The set of core chemistry modules
1219         Co: The set of optional chemistry modules
1220         times: The collection of available time slots.
1221
1222     Returns:
1223         A tuple containing the decision variables x and y.
1224     """
1225     modules = Ac + Ao + Bc + Bo + Cc + Co
1226     x, y = get_variables(modules=modules, times=times)
1227     prob = pulp.LpProblem("ExamScheduling", pulp.LpMinimize)
1228     objective_function = sum([y[day] for day in times])
1229     prob += objective_function
1230
1231     M = 1 / len(modules)
1232     for day in times:
1233         prob += M * sum(x[m][day] for m in modules) <= y[day]
1234         prob += sum([x[mod][day] for mod in Ac + Ao]) <= 1
1235         prob += sum([x[mod][day] for mod in Bc + Bo + Co]) <= 1
1236         prob += sum([x[mod][day] for mod in Bc + Bo + Ac]) <= 1
1237         prob += sum([x[mod][day] for mod in Cc + Co + Bo]) <= 1
1238
1239     for mod in modules:
1240         prob += sum(x[mod][day] for day in times) == 1
1241
1242     prob.solve(pulp.apis.PULP_CBC_CMD(msg=False))
1243     return x, y

```

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

## Python input

```
1244 x, y = get_solution(  
1245     Ac=Ac, Ao=Ao, Bc=Bc, Bo=Bo, Cc=Cc, Co=Co, times=times  
1246 )
```

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

```
1247 print(x[2][5].value())
```

## Python output

```
1248 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

```
1249 print(x[2][9].value())
```

## Python output

```
1250 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

```

1251 def get_schedule(x, y, Ac, Ao, Bc, Bo, Cc, Co, times):
1252     """Returns a human readable schedule corresponding to the
1253     solution of given timetabling problem.
1254
1255     Args:
1256         Ac: The set of core art modules
1257         Ao: The set of optional art modules
1258         Bc: The set of core biology modules
1259         Bo: The set of optional biology modules
1260         Cc: The set of core chemistry modules
1261         Co: The set of optional chemistry modules
1262         times: The collection of available time slots.
1263
1264     Returns:
1265         A string with the schedule
1266     """
1267     modules = Ac + Ao + Bc + Bo + Cc + Co
1268
1269     schedule = ""
1270     for day in times:
1271         if y[day].value() == 1:
1272             schedule += f"\nDay {day}: "
1273             for mod in modules:
1274                 if x[mod][day].value() == 1:
1275                     schedule += f"{mod}, "
1276     return schedule

```

Thus:

## Python input

```

1277 schedule = get_schedule(
1278     x=x,
1279     y=y,
1280     times=times,
1281     Ac=Ac,
1282     Ao=Ao,
1283     Bc=Bc,
1284     Bo=Bo,
1285     Cc=Cc,
1286     Co=Co,
1287 )
1288 print(schedule)

```

gives:

## Python output

```

1289 Day 0: 1, 12,
1290 Day 5: 0, 13,
1291 Day 6: 11,
1292 Day 7: 4, 6, 10,
1293 Day 8: 3, 5, 9,
1294 Day 9: 2, 7,
1295 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 as well as the R `rcbc` package.

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 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

```

1296  #' Writes the row of coefficients for the objective function
1297  #'
1298  #' @param n_modules: the number of modules to schedule
1299  #' @param n_days: the maximum number of days to schedule
1300  #'
1301  #' @return the objective function row to minimise
1302  write_objective <- function(n_modules, n_days){
1303    all_days <- rep(0, n_modules * n_days)
1304    Ys <- rep(1, n_days)
1305    append(all_days, Ys)
1306  }

```

For 3 modules and 3 days:

R input

```

1307  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

```

1308  [1] 0 0 0 0 0 0 0 0 0 1 1 1

```

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

## R input

```

1309 #' Writes the constraint row dealing with clashes
1310 #'
1311 #' @param clashes: a vector of module indices that all cannot
1312 #'                be scheduled at the same time
1313 #' @param day: an integer representing the day
1314 #'
1315 #' @return the constraint row corresponding to that set of
1316 #'         clashes on that day
1317 write_X_clashes <- function(clashes, day, n_days, n_modules){
1318   today <- rep(0, n_modules)
1319   today[clashes] = 1
1320   before_today <- rep(0, n_modules * (day - 1))
1321   after_today <- rep(0, n_modules * (n_days - day))
1322   all_days <- c(before_today, today, after_today)
1323   full_coeffs <- c(all_days, rep(0, n_days))
1324   full_coeffs
1325 }

```

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

```

1326 #' Writes the constraint row to ensure that every module is
1327 #' scheduled once and only one
1328 #'
1329 #' @param module: an integer representing the module
1330 #'
1331 #' @return the constraint row corresponding to scheduling a
1332 #'         module on only one day
1333 write_X_requirements <- function(module, n_days, n_modules){
1334   today <- rep(0, n_modules)
1335   today[module] = 1
1336   all_days <- rep(today, n_days)
1337   full_coeffs <- c(all_days, rep(0, n_days))
1338   full_coeffs
1339 }

```

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

## R input

```

1340 #' Writes the constraint row representing the Y variable,
1341 #' whether at least one exam is scheduled on that day
1342 #'
1343 #' @param day: an integer representing the day
1344 #'
1345 #' @return the constraint row corresponding to creating Y
1346 write_Y_constraints <- function(day, n_days, n_modules){
1347   today <- rep(1, n_modules)
1348   before_today <- rep(0, n_modules * (day - 1))
1349   after_today <- rep(0, n_modules * (n_days - day))
1350   all_days <- c(before_today, today, after_today)
1351   all_Ys <- rep(0, n_days)
1352   all_Ys[day] = -n_modules
1353   full_coeffs <- append(all_days, all_Ys)
1354   full_coeffs
1355 }

```

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

```

1356 #' Writes all the constraints as a matrix, column of
1357 #' inequalities, and right hand side column.
1358 #'
1359 #' @param list_clashes: a list of vectors with sets of modules
1360 #' that cannot be scheduled at the same time
1361 #'
1362 #' @return f.con the LHS of the constraints as a matrix
1363 #' @return f.dir the directions of the inequalities
1364 #' @return f.rhs the values of the RHS of the inequalities
1365 write_constraints <- function(list_clashes, n_days, n_modules){
1366   all_rows <- c()
1367   all_dirs <- c()
1368   all_rhss <- c()
1369   n_rows <- 0
1370
1371   for (clash in list_clashes){
1372     for (day in 1:n_days){
1373       clashes <- write_X_clashes(clash, day, n_days, n_modules)
1374       all_rows <- append(all_rows, clashes)
1375       all_dirs <- append(all_dirs, "<=")
1376       all_rhss <- append(all_rhss, 1)
1377       n_rows <- n_rows + 1
1378     }
1379   }
1380   for (module in 1:n_modules){
1381     reqs <- write_X_requirements(module, n_days, n_modules)
1382     all_rows <- append(all_rows, reqs)
1383     all_dirs <- append(all_dirs, "==")
1384     all_rhss <- append(all_rhss, 1)
1385     n_rows <- n_rows + 1
1386   }
1387   for (day in 1:n_days){
1388     Yconstraints <- write_Y_constraints(day, n_days, n_modules)
1389     all_rows <- append(all_rows, Yconstraints)
1390     all_dirs <- append(all_dirs, "<=")
1391     all_rhss <- append(all_rhss, 0)
1392     n_rows <- n_rows + 1
1393   }
1394   f.con <- matrix(all_rows, nrow = n_rows, byrow = TRUE)
1395   f.dir <- all_dirs
1396   f.rhs <- all_rhss
1397   list(f.con, f.dir, f.rhs)
1398 }

```

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

```

1399 write_constraints(
1400   list_clashes = list(c(1, 2)),
1401   n_days = 2,
1402   n_modules = 2
1403 )

```

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

```

1404 [[1]]
1405      [,1] [,2] [,3] [,4] [,5] [,6]
1406 [1,]    1    1    0    0    0    0
1407 [2,]    0    0    1    1    0    0
1408 [3,]    1    0    1    0    0    0
1409 [4,]    0    1    0    1    0    0
1410 [5,]    1    1    0    0   -2    0
1411 [6,]    0    0    1    1    0   -2
1412
1413 [[2]]
1414 [1] "<=" "<=" "==" "==" "<=" "<="
1415
1416 [[3]]
1417 [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

```

1418 n_modules = 14
1419 n_days = 14
1420 Ac <- c(0, 1)
1421 Ao <- c(2, 3, 4)
1422 Bc <- c(5, 6)
1423 Bo <- c(7, 8)
1424 Cc <- c(9, 10)
1425 Co <- c(11, 12, 13)
1426 list_clashes <- list(
1427   c(Ac, Ao),
1428   c(Bc, Bo, Co),
1429   c(Bc, Bo, Ac),
1430   c(Bo, Cc, Co)
1431 )

```

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

R input

```

1432 constraints <- write_constraints(
1433   list_clashes = list_clashes,
1434   n_days = n_days,
1435   n_modules = n_modules
1436 )
1437 f.con <- constraints[[1]]
1438 f.dir <- constraints[[2]]
1439 f.rhs <- constraints[[3]]
1440 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:

R input

```
1441 library(ROI)
1442
1443 milp <- OP(
1444   objective = L_objective(f.obj),
1445   constraints = L_constraint(
1446     L = f.con,
1447     dir = f.dir,
1448     rhs = f.rhs
1449   ),
1450   types = rep("B", length(f.obj)),
1451   maximum = FALSE
1452 )
```

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

Now to solve:

R input

```
1453 sol <- ROI_solve(milp)
```

The solver will output information about the solve process and runtime.

R input

```
1454 print(sol$solution)
```

```

1455     [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
1456    [30] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1457    [59] 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
1458    [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
1459   [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
1460   [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
1461   [175] 0 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
1462   [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:

```

1463 #' Gives a human readable schedule corresponding to the
1464 #' solution of a given timetable problem.
1465 #'
1466 #' @param sol: a solution to the timetabling problem
1467 #' @param n_modules: the number of modules to schedule
1468 #' @param n_days: the maximum number of days to schedule
1469 #'
1470 #' @return A string with the schedule
1471 get_schedule <- function(sol, n_days, n_modules){
1472   schedule = ""
1473   for (day in 1:n_days){
1474     if (sol$solution[(n_days * n_modules) + day] == 1){
1475       schedule <- paste(schedule, "\n", "Day", day, ":")
1476       for (module in 1:n_modules){
1477         var <- ((day - 1) * n_modules) + module
1478         if (sol$solution[var] == 1){
1479           schedule <- paste(schedule, module)
1480         }
1481       }
1482     }
1483   }
1484   schedule
1485 }

```

Thus:

## R input

```
1486 schedule <- get_schedule(  
1487   sol = sol,  
1488   n_days = n_days,  
1489   n_modules = n_modules  
1490 )  
1491 cat(schedule)
```

gives:

## R output

```
1492 "Day 2 : 4 11"  
1493 "Day 6 : 1 12"  
1494 "Day 8 : 7"  
1495 "Day 10 : 8"  
1496 "Day 11 : 3 13"  
1497 "Day 12 : 2 6 9 14"  
1498 "Day 14 : 5 10"
```

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

## 8.5 RESEARCH

---





# Heuristics

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

## 9.1 PROBLEM

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. Heuristics are a family of methods that can be used to find a *sufficiently good* solution, though not necessarily the optimal solution, where the emphasis is on prioritising computational efficiency.

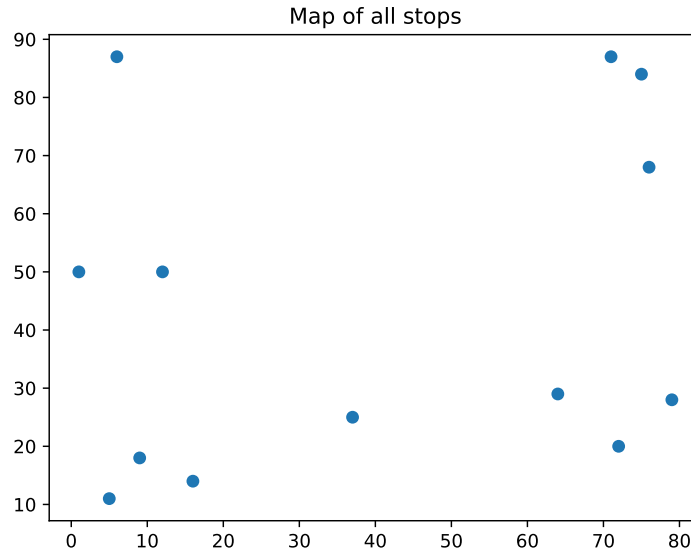


Figure 9.1 The positions of the required stops.

The heuristic approach taken here will be to use a neighbourhood search algorithm. This algorithm works by considering a given potential solution, evaluating it and then trying another potential solution *close* to it. What *close* means depends on different approaches and problems: it is referred to as the neighbourhood. When 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:

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

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

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

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

This is shown diagrammatically in Figure 9.2.

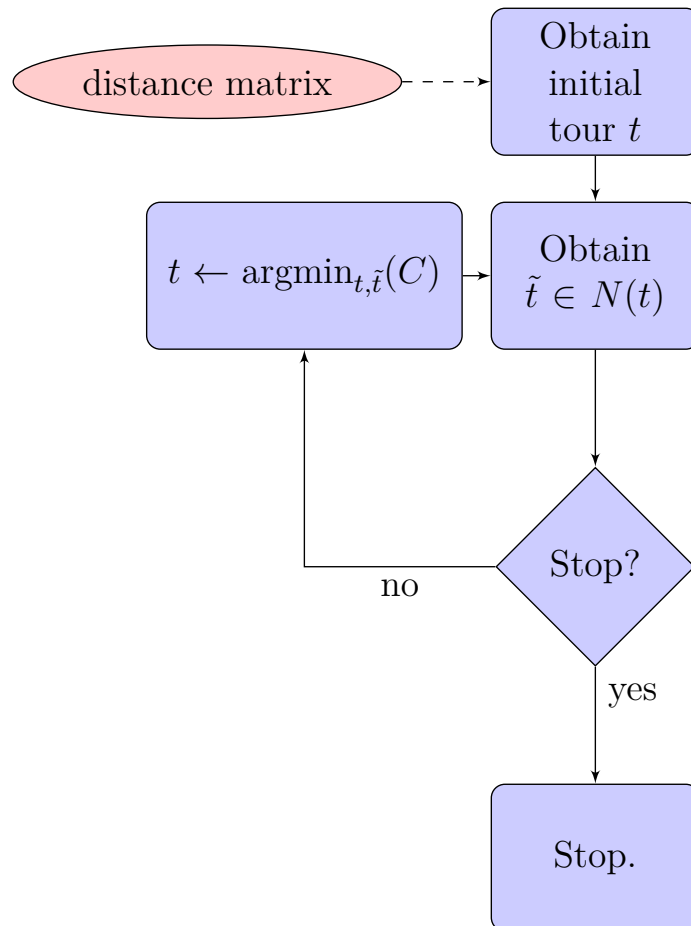


Figure 9.2 The general neighbourhood search algorithm.  $N(t)$  refers to some neighbourhood of  $t$ .

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

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

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

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

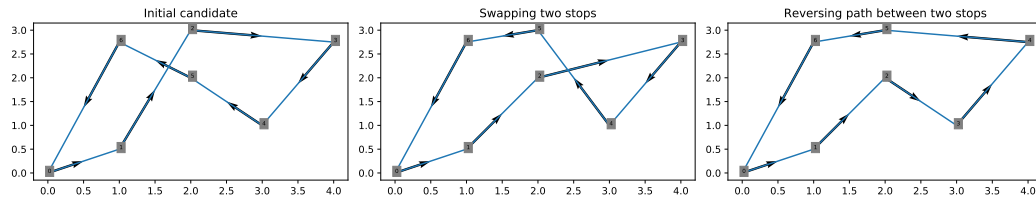
One possible neighbourhood is to choose 2 stops at random and swap. For example, the tour  $\tilde{t}^{(1)} \in N(t)$  is obtained by swapping the 2nd and 5th stops.

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

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

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

Examples of these tours are shown in Figure 9.3.



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

### 9.3 SOLVING WITH PYTHON

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

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

## Python input

```

1499 import numpy as np
1500
1501
1502 def get_initial_candidate(number_of_stops, seed):
1503     """Return an random initial tour.
1504
1505     Args:
1506         number_of_stops: The number of stops
1507         seed: An integer seed.
1508
1509     Returns:
1510         A tour starting an ending at stop with index 0.
1511     """
1512     internal_stops = list(range(1, number_of_stops))
1513     np.random.seed(seed)
1514     np.random.shuffle(internal_stops)
1515     return [0] + internal_stops + [0]

```

This gives a random tour on 13 stops:

## Python input

```

1516 number_of_stops = 13
1517 seed = 0
1518 initial_candidate = get_initial_candidate(
1519     number_of_stops=number_of_stops,
1520     seed=seed,
1521 )
1522 print(initial_candidate)

```

## Python output

```

1523 [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

```
1524 def get_cost(tour, distance_matrix):
1525     """Return the cost of a tour.
1526
1527     Args:
1528         tour: A given tuple of successive stops.
1529         distance_matrix: The distance matrix of the problem.
1530
1531     Returns:
1532         The cost
1533     """
1534     return sum(
1535         distance_matrix[current_stop, next_stop]
1536         for current_stop, next_stop in zip(tour[:-1], tour[1:])
1537     )
```

## Python input

```

1538 distance_matrix = np.array(
1539     (
1540         (0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1541         (35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1542         (35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1543         (29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1544         (70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1545         (35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1546         (42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1547         (27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1548         (24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1549         (44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1550         (58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1551         (71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1552         (69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0),
1553     )
1554 )
1555 cost = get_cost(
1556     tour=initial_candidate,
1557     distance_matrix=distance_matrix,
1558 )
1559 print(cost)

```

## Python output

```

1560 827

```

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

## Python input

```

1561 def swap_stops(tour):
1562     """Return a new tour by swapping two stops.
1563
1564     Args:
1565         tour: A given tuple of successive stops.
1566
1567     Returns:
1568         A tour
1569     """
1570     number_of_stops = len(tour) - 1
1571     i, j = np.random.choice(range(1, number_of_stops), 2)
1572     new_tour = list(tour)
1573     new_tour[i], new_tour[j] = tour[j], tour[i]
1574     return new_tour

```

Applying this neighbourhood operator to the initial candidate gives:

## Python input

```

1575 print(swap_stops(initial_candidate))

```

which swaps the 10th and 12th stops:

## Python output

```

1576 [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

```

1577 def run_neighbourhood_search(
1578     distance_matrix,
1579     iterations,
1580     seed,
1581     neighbourhood_operator=swap_stops,
1582 ):
1583     """Returns a tour by carrying out a neighbourhood search.
1584
1585     Args:
1586         distance_matrix: the distance matrix
1587         iterations: the number of iterations for which to
1588             run the algorithm
1589         seed: a random seed
1590         neighbourhood_operator: the neighbourhood operator
1591             (default: swap_stops)
1592
1593     Returns:
1594         A tour
1595     """
1596     number_of_stops = len(distance_matrix)
1597     candidate = get_initial_candidate(
1598         number_of_stops=number_of_stops,
1599         seed=seed,
1600     )
1601     best_cost = get_cost(
1602         tour=candidate,
1603         distance_matrix=distance_matrix,
1604     )
1605     for _ in range(iterations):
1606         new_candidate = neighbourhood_operator(candidate)
1607         if (
1608             cost := get_cost(
1609                 tour=new_candidate,
1610                 distance_matrix=distance_matrix,
1611             )
1612         ) <= best_cost:
1613             best_cost = cost
1614             candidate = new_candidate
1615
1616     return candidate

```

Now running this for 1000 iterations:

Python input

```

1617 number_of_iterations = 1000
1618
1619 solution_with_swap_stops = run_neighbourhood_search(
1620     distance_matrix=distance_matrix,
1621     iterations=number_of_iterations,
1622     seed=seed,
1623     neighbourhood_operator=swap_stops,
1624 )
1625 print(solution_with_swap_stops)

```

gives:

Python output

```

1626 [0, 7, 2, 8, 5, 3, 1, 9, 12, 11, 4, 10, 6, 0]

```

This has a cost:

Python input

```

1627 cost = get_cost(
1628     tour=solution_with_swap_stops,
1629     distance_matrix=distance_matrix,
1630 )
1631 print(cost)

```

Python output

```

1632 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

```

1633 def reverse_path(tour):
1634     """Return a new tour by reversing the path between two
1635     stops.
1636
1637     Args:
1638         tour: A given tuple of successive stops.
1639
1640     Returns:
1641         A tour
1642     """
1643     number_of_stops = len(tour) - 1
1644     stops = np.random.choice(range(1, number_of_stops), 2)
1645     i, j = sorted(stops)
1646     new_tour = tour[:i] + tour[i : j + 1][::-1] + tour[j + 1 :]
1647     return new_tour

```

Applying this neighbourhood operator to the initial candidate gives:

Python input

```

1648 print(reverse_path(initial_candidate))

```

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

Python output

```

1649 [0, 7, 4, 10, 8, 2, 9, 3, 11, 5, 12, 1, 6, 0]

```

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

## Python input

```

1650 solution_with_reverse_path = run_neighbourhood_search(
1651     distance_matrix=distance_matrix,
1652     iterations=number_of_iterations,
1653     seed=seed,
1654     neighbourhood_operator=reverse_path,
1655 )
1656 print(solution_with_reverse_path)

```

gives:

## Python output

```

1657 [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

```

1658 cost = get_cost(
1659     tour=solution_with_reverse_path,
1660     distance_matrix=distance_matrix,
1661 )
1662 print(cost)

```

which gives:

## Python output

```

1663 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.

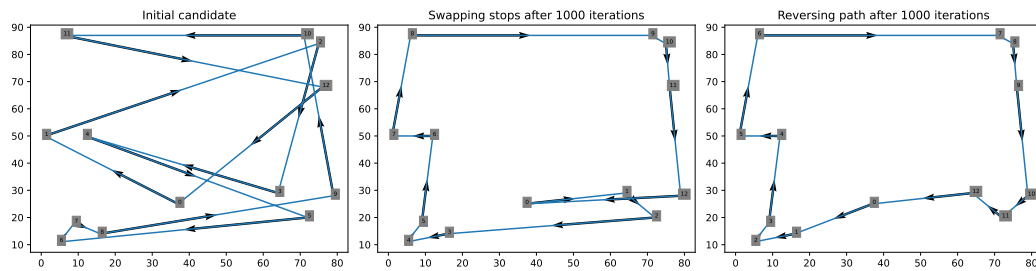


Figure 9.4 The final tours obtained by using the neighbourhood search in Python.

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

R input

```

1664 #' Return an random initial tour.
1665 #'
1666 #' @param number_of_stops The number of stops.
1667 #' @param seed An integer seed.
1668 #'
1669 #' @return A tour starting an ending at stop with index 0.
1670 get_initial_candidate <- function(number_of_stops, seed){
1671   internal_stops <- 1:(number_of_stops - 1)
1672   set.seed(seed)
1673   internal_stops <- sample(internal_stops)
1674   c(0, internal_stops, 0)
1675 }

```

This gives a random tour on 13 stops:

R input

```

1676 number_of_stops <- 13
1677 seed <- 1
1678 initial_candidate <- get_initial_candidate(
1679   number_of_stops = number_of_stops,
1680   seed = seed)
1681 print(initial_candidate)

```

## R output

1682

```
[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

1683

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

1684

```
##
```

1685

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

1686

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

1687

```
##
```

1688

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

1689

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

1690

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

1691

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

1692

```
}
```

## R input

```

1693 distance_matrix <- rbind(
1694     c(0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
1695     c(35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
1696     c(35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
1697     c(29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
1698     c(70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
1699     c(35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
1700     c(42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
1701     c(27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
1702     c(24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
1703     c(44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
1704     c(58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
1705     c(71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
1706     c(69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0)
1707 )
1708 cost <- get_cost(
1709     tour = initial_candidate,
1710     distance_matrix = distance_matrix)
1711 print(cost)

```

## R output

```

1712 [1] 709

```

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

R input

```

1713 #' Return a new tour by swapping two stops.
1714 #'
1715 #' @param tour A given vector of successive stops.
1716 #'
1717 #' @return A tour
1718 swap_stops <- function(tour){
1719   number_of_stops <- length(tour) - 1
1720   stops_to_swap <- sample(2:number_of_stops, 2)
1721   new_tour <- replace(
1722     x = tour,
1723     list = stops_to_swap,
1724     values = rev(tour[stops_to_swap])
1725   )
1726 }

```

Applying this neighbourhood operator to the initial candidate gives:

R input

```

1727 print(swap_stops(initial_candidate))

```

which swaps the 6th and 11th stops:

R output

```

1728 [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

```

1729 #' Returns a tour by carrying out a neighbourhood search
1730 #'
1731 #' @param distance_matrix: the distance matrix
1732 #' @param iterations: the number of iterations for
1733 #'                      which to run the algorithm
1734 #' @param seed: a random seed (default: None)
1735 #' @param neighbourhood_operator: the neighbourhood operation
1736 #'                                (default: swap_stops)
1737 #'
1738 #' @return A tour
1739 run_neighbourhood_search <- function(
1740   distance_matrix,
1741   iterations,
1742   seed = NA,
1743   neighbourhood_operator = swap_stops
1744 ){
1745   number_of_stops <- nrow(distance_matrix)
1746   candidate <- get_initial_candidate(
1747     number_of_stops = number_of_stops,
1748     seed = seed
1749   )
1750   best_cost <- get_cost(
1751     tour = candidate,
1752     distance_matrix = distance_matrix
1753   )
1754   for (repetition in 1:iterations) {
1755     new_candidate <- neighbourhood_operator(candidate)
1756     cost <- get_cost(
1757       tour = new_candidate,
1758       distance_matrix = distance_matrix
1759     )
1760     if (cost <= best_cost) {
1761       best_cost <- cost
1762       candidate <- new_candidate
1763     }
1764   }
1765   candidate
1766 }

```

Now running this for 1000 iterations:

## R input

```

1767 number_of_iterations <- 1000
1768 solution_with_swap_stops <- run_neighbourhood_search(
1769   distance_matrix = distance_matrix,
1770   iterations = number_of_iterations,
1771   seed = seed,
1772   neighbourhood_operator = swap_stops
1773 )
1774 print(solution_with_swap_stops)

```

gives:

## R output

```

1775 [1] 0 11 4 10 6 2 7 12 9 1 3 5 8 0

```

This has a cost:

## R input

```

1776 cost <- get_cost(
1777   tour = solution_with_swap_stops,
1778   distance_matrix = distance_matrix
1779 )
1780 print(cost)

```

which gives:

## R output

```

1781 [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

```

1782 #' Return a new tour by reversing the path between two stops.
1783 #'
1784 #' @param tour A given vector of successive stops.
1785 #'
1786 #' @return A tour
1787 reverse_path <- function(tour){
1788   number_of_stops <- length(tour) - 1
1789   stops_to_swap <- sample(2:number_of_stops, 2)
1790   i <- min(stops_to_swap)
1791   j <- max(stops_to_swap)
1792   new_order <- c(
1793     c(1: (i - 1)),
1794     c(j:i),
1795     c( (j + 1): length(tour))
1796   )
1797   tour[new_order]
1798 }

```

Applying this neighbourhood operator to the initial candidate gives:

## R input

```

1799 print(reverse_path(initial_candidate))

```

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

## R output

```

1800 [1] 0 9 10 12 11 6 8 3 5 2 1 7 4 0

```

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

## R input

```

1801 number_of_iterations <- 1000
1802 solution_with_reverse_path <- run_neighbourhood_search(
1803   distance_matrix = distance_matrix,
1804   iterations = number_of_iterations,
1805   seed = seed,
1806   neighbourhood_operator = reverse_path
1807 )
1808 print(solution_with_reverse_path)

```

gives:

## R output

```

1809 [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

```

1810 cost <- get_cost(
1811   tour = solution_with_reverse_path,
1812   distance_matrix = distance_matrix
1813 )
1814 print(cost)

```

which gives:

## R output

```

1815 [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 RESEARCH

TBA

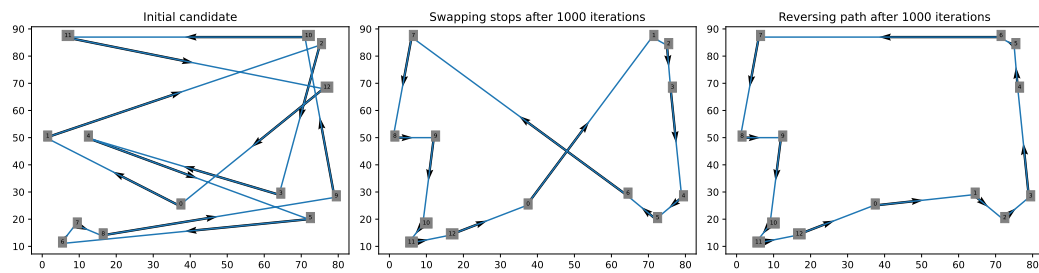


Figure 9.5 The final tours obtained by using the neighbourhood search in R



---

# Bibliography

---

