Half Title Page

Title Page

LOC Page

Vince: to Riggins
Geraint: also, to Riggins

Contents

Foreword

This is the foreword

Preface

This is the preface.

Contributors

Michaél Aftosmis
NASA Ames Research Center
Moffett Field, California

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

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

Gabrielle Allen
Louisiana State University
Baton Rouge, Louisiana

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

Steven F. Ashby
Lawrence Livermore National Laboratory
Livermore, California

David A. Bader
Georgia Institute of Technology
Atlanta, Georgia

Benjamin Bergen
Los Alamos National Laboratory
Los Alamos, New Mexico

Jonathan W. Berry
Sandia National Laboratories
Albuquerque, New Mexico

Martin Berzins
University of Utah
Salt Lake City, Utah

Abhinav Bhatele
University of Illinois
Urbana-Champaign, Illinois

Christian Bischof
RWTH Aachen University
Germany

Rupak Biswas
NASA Ames Research Center
Moffett Field, California

Eric Bohm
University of Illinois
Urbana-Champaign, Illinois

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

Geörge Bosilca
University of Tennessee
Knoxville, Tennessee

Grèg L. Bryan
Columbia University
New York, New York

Marian Bubak
AGH University of Science and Technology
Kraków, Poland

Andrew Canning
Lawrence Berkeley National Laboratory
Berkeley, California

Jonathan Carter
Lawrence Berkeley National Laboratory
Berkeley, California

Zizhong Chen
Jacksonville State University
Jacksonville, Alabama

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

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

I

Getting Started

CHAPTER 1

Introduction

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

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

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

1.1 WHO IS THIS BOOK FOR?

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

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

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

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

- Have access to a computer and be able to connect to the internet (at least once) to be able to download the relevant software.
- Be prepared to read some mathematics. Technically you do not need to understand the specific mathematics to be able to use the tools in this book. The topics covered use some algebra, calculus and probability.

1.2 WHAT DO WE MEAN BY APPLIED MATHEMATICS?

We consider this book to be a book on applied mathematics. This is not however a universal term, for some applied mathematics is the study of mechanics and involves modelling projectiles being fired out of canons. We will use the term a bit more freely here and mean any type of real world problem that can be tackled using mathematical tools. This is sometimes referred to as operational research, operations research, mathematical modelling or indeed just mathematics.

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

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

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

1.3 WHAT IS OPEN SOURCE SOFTWARE

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

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

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

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

1.4 HOW TO GET THE MOST OUT OF THIS BOOK

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

We feel that you can choose to read the book from cover to cover, writing out the code examples as you go; or it could also be used as a reference text when faced with particular problem and wanting to know where to start.

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

Every chapter has the following structure:

1. Introduction - a brief overview of a given problem type. Here we will describe the problem at hand in general terms.
2. An Example problem. This will provide a tangible example problem that offers the reader some intuition for the rest of the discussion.
3. Solving with Python. We will describe the mathematical tools available to us in a programming language called Python to solve the problem.
4. Solving with R. Here we will do the same with the R programming language.
5. Brief theoretic background with pointers to reference texts. Some readers might like to delve in to the mathematics of the problem a bit further, we will include those details here.
6. Examples of research using these methods. Finally, some readers might even be interested in finding out a bit more of what mathematicians are doing on these problems. Often this will include some descriptions of the problem considered but perhaps at a much larger scale than the one presented in the example.

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

II

Probabilistic Modelling

CHAPTER 2

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

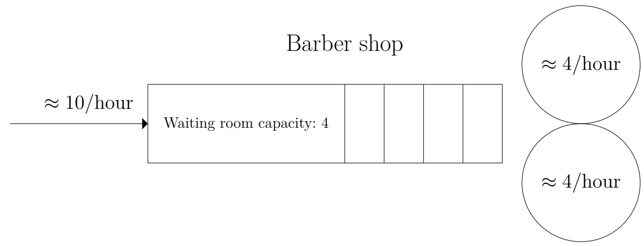


Figure 2.1 Diagrammatic representation of the barber shop as a queuing system.

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:

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

The state increases when people arrive and this happens at a rate of change of 10. The state decrease 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

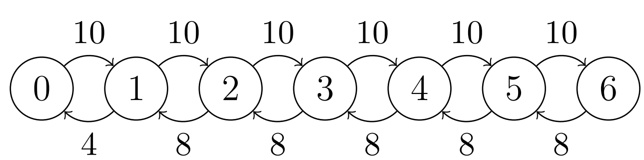


Figure 2.2 Diagrammatic representation of the state space and the transition rates

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:

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

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

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

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

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

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

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

with the following constraint:

$$\sum_{i=1}^{n} \pi_i = 1 \tag{2.5}$$

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

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

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

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

    capacity = waiting_room + num_barbers
    delta = out_state - in_state

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

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

    return 0
```

Next, 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

```python
import itertools
import numpy as np


def get_transition_rate_matrix(waiting_room=4, num_barbers=2):
    """Return the transition matrix Q.

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

    Returns:
        A matrix.
    """
    capacity = waiting_room + num_barbers
    state_pairs = itertools.product(
        range(capacity + 1), repeat=2
    )

    flat_transition_rates = [
        get_transition_rate(
            in_state=in_state,
            out_state=out_state,
            waiting_room=waiting_room,
            num_barbers=num_barbers,
        )
        for in_state, out_state in state_pairs
    ]
    transition_rates = np.reshape(
        flat_transition_rates, (capacity + 1, capacity + 1)
    )
    np.fill_diagonal(
        transition_rates, -transition_rates.sum(axis=1)
    )

    return transition_rates
```

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

Python input

```
Q = get_transition_rate_matrix()
print(Q)
```

which gives:

Python output

```
[[-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]]
```

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

```
import scipy.linalg

print(scipy.linalg.expm(Q * 0.5).round(5))
```

which gives:

Python output

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

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

which gives:

Python output

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

```
def get_steady_state_vector(Q):
    """Return the steady state vector of any given continuous
    time transition rate matrix.

    Args:
        Q: a transition rate matrix

    Returns:
        A vector
    """
    state_space_size, _ = Q.shape
    A = np.vstack((Q.T, np.ones(state_space_size)))
    b = np.append(np.zeros(state_space_size), 1)
    x, _, _, _ = np.linalg.lstsq(A, b, rcond=None)
    return x
```

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

Python input

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

giving:

Python output

```
[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

```python
def get_probability_of_full_shop(
    waiting_room=4, num_barbers=2
):
    """Return the probability of the barber shop being full.

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

    Returns:
        A real.
    """
    Q = get_transition_rate_matrix(
        waiting_room=waiting_room,
        num_barbers=num_barbers,
    )
    pi = get_steady_state_vector(Q)
    return pi[-1]
```

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

Python input

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

which gives:

Python output

```
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

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

which gives:

Python output

```
0.23557
```

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

Python input

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

Python output

```
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

```
#' Return the transition rate for 2 given states.
#'
#' @param in_state an integer
#' @param out_state an integer
#' @param waiting_room an integer (default: 4)
#' @param num_barbers an integer  (default: 2)
#'
#' @return A real
get_transition_rate <- function(in_state,
                                out_state,
                                waiting_room = 4,
                                num_barbers = 2){
  arrival_rate <- 10
  service_rate <- 4

  capacity <- waiting_room + num_barbers
  delta <- out_state - in_state

  if (delta == 1) {
    if (in_state < capacity) {
      return(arrival_rate)
    }
  }

  if (delta == -1) {
    return(min(in_state, num_barbers) * service_rate)
  }
  return(0)
}
```

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

R input

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

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

R input

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

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

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

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

R input

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

which gives:

R output

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]  -10   10    0    0    0    0    0
[2,]    4  -14   10    0    0    0    0
[3,]    0    8  -18   10    0    0    0
[4,]    0    0    8  -18   10    0    0
[5,]    0    0    0    8  -18   10    0
[6,]    0    0    0    0    8  -18   10
[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

```
library(expm, warn.conflicts = FALSE, quietly = TRUE)
library(magrittr, warn.conflicts = FALSE, quietly = TRUE)

print( (Q * .5) %>% expm %>% round(5))
```

which gives:

R output

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

After 500 time units:

R input

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

which gives:

R output

```
        [,1]    [,2]    [,3]    [,4]    [,5]   [,6]    [,7]
[1,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
[2,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
[3,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
[4,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
[5,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
[6,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
[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

```
library(pracma, warn.conflicts = FALSE, quietly = TRUE)

#' Return the steady state vector of any given continuous time
#' transition rate matrix
#'
#' @param Q a transition rate matrix
#'
#' @return A vector
get_steady_state_vector <- function(Q){
  state_space_size <- dim(Q)[1]
  A <- rbind(t(Q), 1)
  b <- c(integer(state_space_size), 1)
  mldivide(A, b)
}
```

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

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

giving:

R output

```
           [,1]
[1,] 0.03430888
[2,] 0.08577220
[3,] 0.10721525
[4,] 0.13401906
[5,] 0.16752383
[6,] 0.20940479
[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

```r
#' Return the probability of the barber shop being full
#'
#' @param waiting_room (default: 4)
#' @param num_barbers (default: 2)
#'
#' @return A real
get_probability_of_full_shop <- function(waiting_room = 4,
                                         num_barbers = 2){
  arrival_rate <- 10
  service_rate <- 4
  pi <- get_transition_rate_matrix(
    waiting_room = waiting_room,
    num_barbers = num_barbers
    ) %>%
    get_steady_state_vector()

  capacity <- waiting_room + num_barbers
  pi[capacity + 1]
}
```

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

R input

```r
print(get_probability_of_full_shop())
```

which gives:

R output

```
[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

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

which decreases the probability of a full shop to:

R output

```
[1] 0.2355699
```

but adding another barber and chair:

R input

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

gives:

R output

```
[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

CHAPTER 3

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?

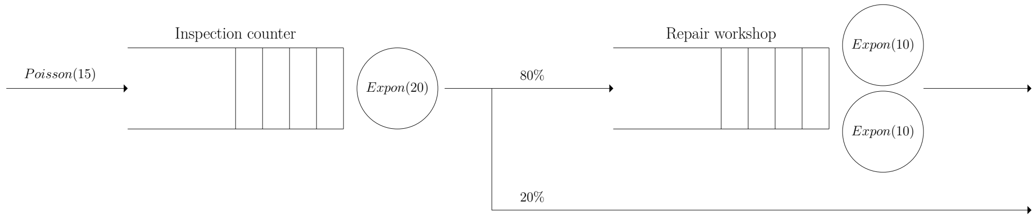


Figure 3.1 Diagrammatic representation of the bicycle repair shop as a queuing system.

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:

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

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

```python
import ciw


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

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

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

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

Python input

```
N = build_network_object()
print(N.number_of_nodes)
```

which gives:

Python output

```
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

```
def run_simulation(network, seed=0):
    """Builds a simulation object and runs it for 8 time units.

    Args:
        network: a Ciw network object
        seed: a float (default: 0)

    Returns:
        a Ciw simulation object after a run of the simulation
    """
    max_time = 8
    ciw.seed(seed)
    Q = ciw.Simulation(network)
    Q.simulate_until_max_time(max_time)
    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

```python
import pandas as pd


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

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

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

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

Python input

```python
N = build_network_object()
Q = run_simulation(N)
p = get_proportion(Q)
print(round(p, 6))
```

This gives:

Python output

```
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

```python
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

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

which gives:

Python output

```
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

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

which gives:

Python output

```
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

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

which gives:

Python output

```
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 than a bicycle would take at the repair shop:

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

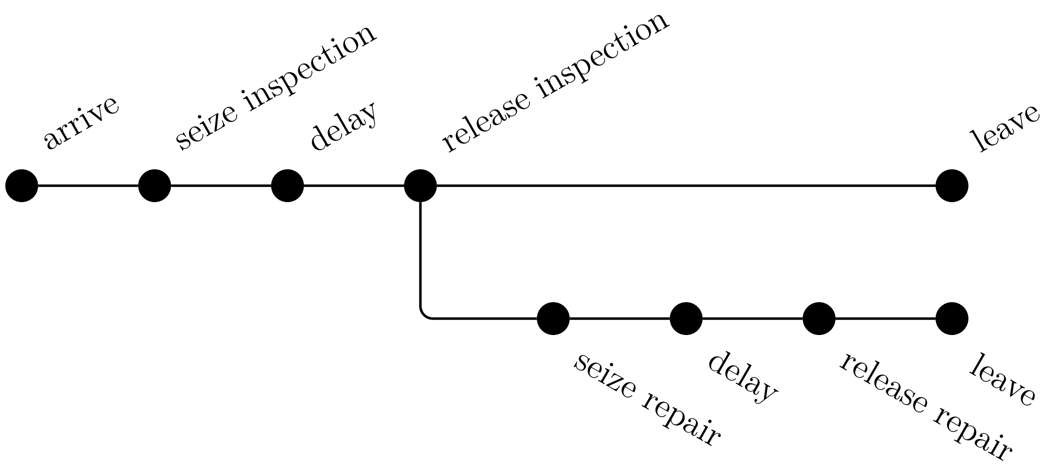


Figure 3.2 Diagrammatic representation of the forked trajectories a bicycle can take

R input

```r
library(simmer)

#' Returns a simmer trajectory object outlining the bicycles
#' path through the repair shop
#'
#' @return A simmer trajectory object
define_bicycle_trajectories <- function() {
  inspection_rate <- 20
  repair_rate <- 10
  prob_need_repair <- 0.8
  bicycle <-
    trajectory("Inspection") %>%
    seize("Inspector") %>%
    timeout(function() {
      rexp(1, inspection_rate)
    }) %>%
    release("Inspector") %>%
    branch(
      function() (runif(1) < prob_need_repair),
      continue = c(F),
      trajectory("Repair") %>%
        seize("Repairer") %>%
        timeout(function() {
          rexp(1, repair_rate)
        }) %>%
        release("Repairer"),
      trajectory("Out")
    )
  return(bicycle)
}
```

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

```r
#' Runs one trial of the simulation.
#'
#' @param bicycle a simmer trajectory object
#' @param num_inspectors positive integer (default: 1)
#' @param num_repairers positive integer (default: 2)
#' @param seed a float (default: 0)
#'
#' @return A simmer simulation object after one run of
#'         the simulation
run_simulation <- function(bicycle,
                           num_inspectors = 1,
                           num_repairers = 2,
                           seed = 0) {
  arrival_rate <- 15
  max_time <- 8
  repair_shop <-
    simmer("Repair Shop") %>%
    add_resource("Inspector", num_inspectors) %>%
    add_resource("Repairer", num_repairers) %>%
    add_generator("Bicycle", bicycle, function() {
      rexp(1, arrival_rate)
    })

  set.seed(seed)
  repair_shop %>% run(until = 8)
  return(repair_shop)
}
```

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

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

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

R input

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

This piece of code gives

R output

```
[1] 0.1343284
```

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

```
#' Returns the average proportion of bicycles spending over
#' a given limit at the repair shop.
#'
#' @param num_inspectors positive integer (default: 1)
#' @param num_repairers positive integer (default: 2)

#' @return a float between 0 and 1
get_average_proportion <- function(num_inspectors = 1,
                                   num_repairers = 2) {
  num_trials <- 100
  bicycle <- define_bicycle_trajectories()
  proportions <- c()
  for (trial in 1:num_trials) {
    repair_shop <- run_simulation(
      bicycle = bicycle,
      num_inspectors = num_inspectors,
      num_repairers = num_repairers,
      seed = trial
    )
    proportion <- get_proportion(
      repair_shop = repair_shop
    )
    proportions[trial] <- proportion
  }
  return(mean(proportions))
}
```

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

R input

```
print(
  get_average_proportion(
    num_inspectors = 1,
    num_repairers = 2)
)
```

which gives:

R output

```
[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

```
print(
  get_average_proportion(
    num_inspectors = 2,
    num_repairers = 2)
)
```

which gives:

R output

```
[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

```
print(
  get_average_proportion(
    num_inspectors = 1,
    num_repairers = 3)
)
```

which gives:

R output

```
[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

CHAPTER 4

Modelling with Differential Equations

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

4.1 PROBLEM

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

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

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 by will be denoted by α, then the differential equation that describes the above situation is:

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

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

In this particular case, one such function is:

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

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:

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

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:

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

where K is the cost per person per unit time.

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

4.3 SOLVING WITH PYTHON

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

Python input

```python
import sympy as sym

t = sym.Symbol("t")
alpha = sym.Symbol("alpha")
I_0 = sym.Symbol("I_0")
I = sym.Function("I")


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

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

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

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

Python input

```python
eq = get_equation()
print(eq)
```

which gives:

Python output

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

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

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

A value of α can be passed if required:

Python input

```python
eq = get_equation(alpha=1)
print(eq)
```

Python output

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

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

Python input

```python
def get_solution(I_0=I_0, alpha=alpha):
    """Return the solution to the differential equation.

    Args:
        I_0: a float (default: symbolic I_0)
        alpha: a float (default: symbolic alpha)

    Returns:
        A symbolic equation
    """
    eq = get_equation(alpha=alpha)
    return sym.dsolve(eq, I(t), ics={I(0): I_0})
```

This can verify the solution discussed previously:

Python input

```python
sol = get_solution()
print(sol)
```

which gives:

Python output

```
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

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

which gives:

Python output

```
True
```

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

Python input

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

which gives:

Python output

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

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

Python input

```
def get_cost(
    I_0=I_0,
    alpha=alpha,
    cost_per_person=10,
    cost_of_cure=0,
):
    """Return the cost.

    Args:
        I_0: a float (default: symbolic I_0)
        alpha: a float (default: symbolic alpha)
        cost_per_person: a float (default: 10)
        cost_of_cure: a float (default: 0)

    Returns:
        A symbolic expression
    """
    I_sol = get_solution(I_0=I_0, alpha=alpha)
    return (
        sym.integrate(I_sol.rhs, (t, 0, sym.oo))
        * cost_per_person
        + cost_of_cure * I_0
    )
```

The cost without purchasing the cure is:

Python input

```
I_0 = 100
alpha = 2
cost_without_cure = get_cost(I_0=I_0, alpha=alpha)
print(cost_without_cure)
```

which gives:

Python output

```
500
```

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

Python input

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

which gives:

Python output

```
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

```
derivative <- function(t, y, parameters) {
  with(as.list(c(y, parameters)), {
    dIdt <- -alpha * I  # nolint
    list(dIdt)  # nolint
  })
}
```

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

R input

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

This gives:

R output

```
[[1]]
[1] -200
```

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

R input

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

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

```
get_cost <- function(
                     I_0 = 100,
                     alpha = 2,
                     cost_per_person = 10,
                     cost_of_cure = 0,
                     step_size = 0.0001,
                     max_time = 10) {
  times <- seq(0, max_time, by = step_size)
  out <- integrate_ode(times,
    y0 = c(I = I_0),
    alpha = alpha
  )
  number_of_observations <- length(out[, "I"])

  stopifnot(out[number_of_observations, "I"] < step_size)

  time_between_steps <- diff(out[, "time"])
  area_under_curve <- sum(
    time_between_steps *
      out[-number_of_observations, "I"]
  )
  area_under_curve *
    cost_per_person + cost_of_cure *
      I_0
}
```

Note that this function uses `stopifnot` to make sure the differential equation has been solved for a long enough time period.

The cost without purchasing the cure is:

R input

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

which gives:

R output

```
[1] 500
```

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

R input

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

which gives:

R output

```
[1] 750
```

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

4.5 RESEARCH

TBA

CHAPTER 5

Systems Dynamics

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

5.1 PROBLEM

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

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

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

5.2 THEORY

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

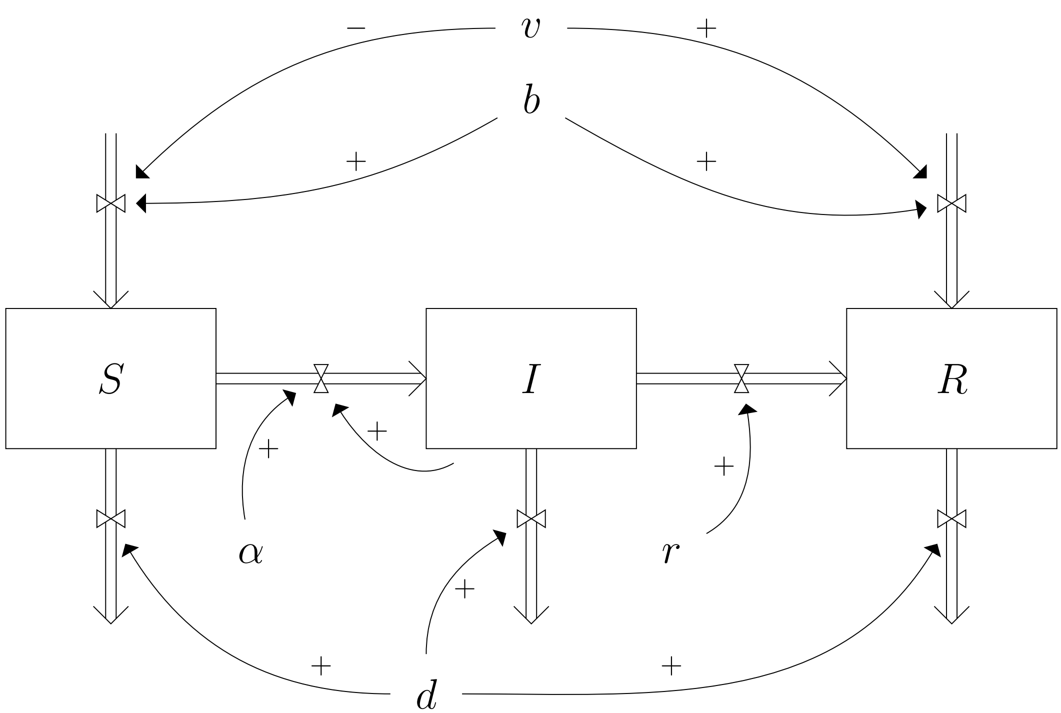


Figure 5.1 Diagrammatic representation of the epidemiology model

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

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

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

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

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

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

$$\frac{dR}{dt} = rI - dR + vbN \tag{5.3}$$

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

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

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

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

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

5.3 SOLVING WITH PYTHON

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

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

Python input

```python
def derivatives(y, t, vaccine_rate, birth_rate=0.01):
    """Defines the system of differential equations that
    describe the epidemiology model.

    Args:
        y: a tuple of three integers
        t: a positive float
        vaccine_rate: a positive float <= 1
        birth_rate: a positive float <= 1

    Returns:
        A tuple containing dS, dI, and dR
    """
    infection_rate = 0.3
    recovery_rate = 0.02
    death_rate = 0.01
    S, I, R = y
    N = S + I + R
    dSdt = (
        -((infection_rate * S * I) / N)
        + ((1 - vaccine_rate) * birth_rate * N)
        - (death_rate * S)
    )
    dIdt = (
        ((infection_rate * S * I) / N)
        - (recovery_rate * I)
        - (death_rate * I)
    )
    dRdt = (
        (recovery_rate * I)
        - (death_rate * R)
        + (vaccine_rate * birth_rate * N)
    )
    return dSdt, dIdt, dRdt
```

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

Python input

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

Python output

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

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

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

Python input

```python
from scipy.integrate import odeint


def integrate_ode(
    derivative_function,
    t,
    y0=(2999, 1, 0),
    vaccine_rate=0.85,
    birth_rate=0.01,
):
    """Numerically solve the system of differential equations.

    Args:
        derivative_function: a function returning a tuple
                             of three floats
        t: an array of increasing positive floats
        y0: a tuple of three integers (default: (2999, 1, 0))
        vaccine_rate: a positive float <= 1 (default: 0.85)
        birth_rate: a positive float <= 1 (default: 0.01)

    Returns:
        A tuple of three arrays
    """
    results = odeint(
        derivative_function,
        y0,
        t,
        args=(vaccine_rate, birth_rate),
    )
    S, I, R = results.T
    return S, I, R
```

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

Begin with a vaccine rate of 0%:

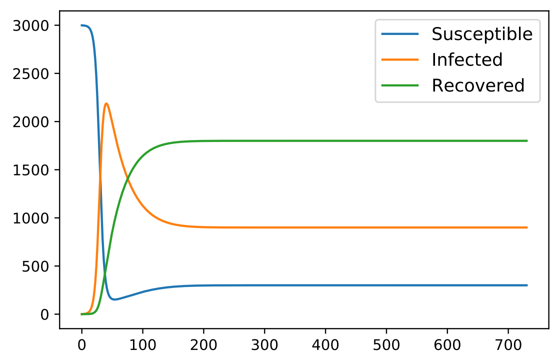


Figure 5.2 Output of code line 737-742

Python input

```python
import numpy as np
from scipy.integrate import odeint

t = np.arange(0, 730.01, 0.01)
S, I, R = integrate_ode(derivatives, t, vaccine_rate=0.0)
```

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

Python input

```python
import matplotlib.pyplot as plt

fig, ax = plt.subplots(1)
ax.plot(t, S, label='Susceptible')
ax.plot(t, I, label='Infected')
ax.plot(t, R, label='Recovered')
ax.legend(fontsize=12)
fig.savefig("plot_no_vaccine_python.pdf")
```

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

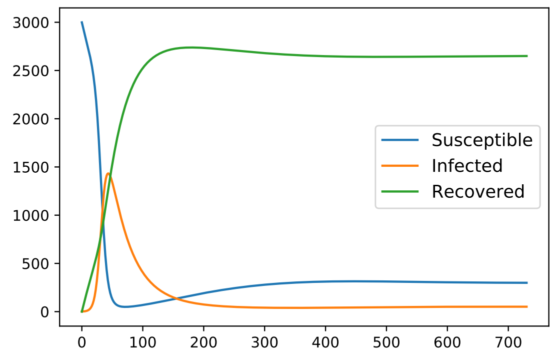


Figure 5.3 Output of code line 745-750

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

Now with a vaccine rate of 85%:

Python input

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

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

Python input

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

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

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

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

Python input

```python
def daily_cost(
    derivative_function=derivatives, vaccine_rate=0.85
):
    """Calculates the daily cost to the public health system
    after 2 years.

    Args:
        derivative_function: a function returning a tuple
                              of three floats
        vaccine_rate: a positive float <= 1 (default: 0.85)

    Returns:
        the daily cost
    """
    max_time = 730
    time_step = 0.01
    birth_rate = 0.01
    vaccine_cost = 220
    medication_cost = 10
    t = np.arange(0, max_time + time_step, time_step)
    S, I, R = integrate_ode(
        derivatives,
        t,
        vaccine_rate=vaccine_rate,
        birth_rate=birth_rate,
    )
    N = S[-1] + I[-1] + R[-1]
    daily_vaccine_cost = (
        N * birth_rate * vaccine_rate * vaccine_cost
    ) / time_step
    daily_meds_cost = (I[-1] * medication_cost) / time_step
    return daily_vaccine_cost + daily_meds_cost
```

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

Python input

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

which gives

Python output

```
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

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

which gives

Python output

```
611903.36
```

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

5.4 SOLVING WITH R

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

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

R input

```r
#' Defines the system of differential equations that describe
#' the epidemiology model.
#'
#' @param t a positive float
#' @param y a tuple of three integers
#' @param vaccine_rate a positive float <= 1
#' @param birth_rate a positive float <= 1
#'
#' @return a list containing dS, dI, and dR
derivatives <- function(t, y, parameters){
  infection_rate <- 0.3
  recovery_rate <- 0.02
  death_rate <- 0.01
  with(as.list(c(y, parameters)), {
    N <- S + I + R
    dSdt <- ( - ( (infection_rate * S * I) / N)  # nolint
             + ( (1 - vaccine_rate) * birth_rate * N)
             - (death_rate * S))
    dIdt <- ( ( (infection_rate * S * I) / N)  # nolint
            - (recovery_rate * I)
            - (death_rate * I))
    dRdt <- ( (recovery_rate * I)  # nolint
             - (death_rate * R)
             + (vaccine_rate * birth_rate * N))
    list(c(dSdt, dIdt, dRdt))  # nolint
  })
}
```

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

R input

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

R output

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

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

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

R input

```
library(deSolve)  # nolint

#' Numerically solve the system of differential equations
#'
#' @param t an array of increasing positive floats
#' @param y0 list of integers (default: c(S=2999, I=1, R=0))
#' @param birth_rate a positive float <= 1 (default: 0.01)
#' @param vaccine_rate a positive float <= 1 (default: 0.85)
#'
#' @return a matrix of times, S, I and R values
integrate_ode <- function(times,
                          y0 = c(S = 2999, I = 1, R = 0),
                          birth_rate = 0.01,
                          vaccine_rate = 0.84){
  params <- c(birth_rate = birth_rate,
              vaccine_rate = vaccine_rate)
  ode(y = y0,
      times = times,
      func = derivatives,
      parms = params)
}
```

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

Begin with a vaccine rate of 0%:

R input

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

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

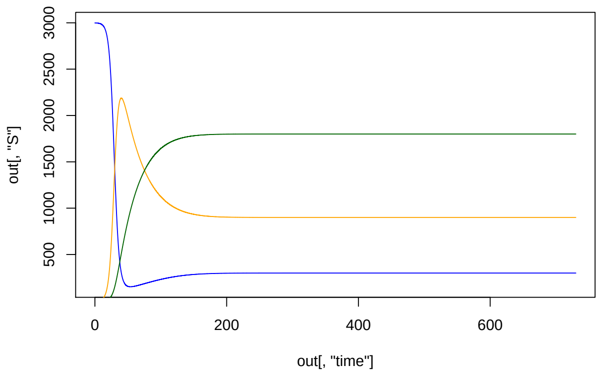


Figure 5.4 Output of code line 846-850

R input

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

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

Now with a vaccine rate of 85%:

R input

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

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

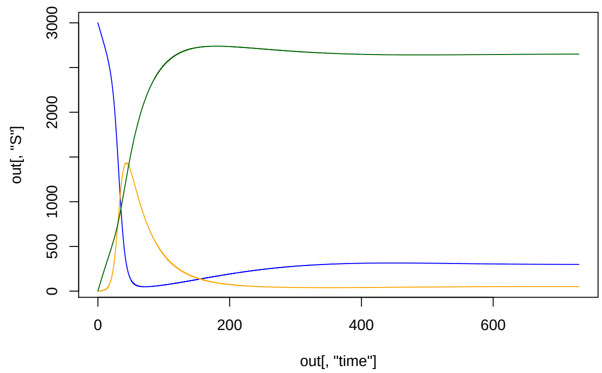


Figure 5.5 Output of code line 853-857

R input

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

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

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

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

R input

```r
#' Calculates the daily cost to the public health
#' system after 2 years
#'
#' @param derivative_function: a function returning a
#'                              list of three floats
#' @param vaccine_rate: a positive float <= 1 (default: 0.85)
#'
#' @return the daily cost
daily_cost <- function(derivative_function = derivatives,
                       vaccine_rate = 0.85){
  max_time <- 730
  time_step <- 0.01
  birth_rate <- 0.01
  vaccine_cost <- 220
  medication_cost <- 10
  times <- seq(0, max_time, by = time_step)
  out <- integrate_ode(times, vaccine_rate = vaccine_rate)
  N <- sum(tail(out[, c("S", "I", "R")], n = 1))
  daily_vaccine_cost <- (N
                         * birth_rate
                         * vaccine_rate
                         * vaccine_cost) / time_step
  daily_medication_cost <- ( (tail(out[, "I"], n = 1)
                              * medication_cost)) / time_step
  daily_vaccine_cost + daily_medication_cost
}
```

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

R input

```r
cost <- daily_cost(vaccine_rate = 0.0)
print(cost)
```

which gives

R output

```
[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

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

which gives

R output

```
[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

CHAPTER 6

Game Theory

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

6.1 PROBLEM

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

Some exploratory data analysis gives the following insights:

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

The expected profits are given in Table 6.1.

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

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

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

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

6.2 THEORY

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

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

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

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

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

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

A strategy corresponds to an way of choosing actions, this is represented by a probability vector. For the ith 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 ith action.

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

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

A diagram of the system is shown in Figure 6.1

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

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

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

2 players $\Longrightarrow$

- 2 action sets A_1, A_2;
- 2 payoff functions, represented by matrices M, N.

$$
\begin{array}{c c}
 & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\
\begin{array}{l} \text{1 taxi} \\ \text{2 taxis} \\ \text{3 taxis} \end{array} & M = \begin{pmatrix} 1 & 1/2 & 1/3 \\ 3/2 & 19/20 & 1/2 \\ 5/3 & 4/5 & 17/20 \end{pmatrix}
\end{array}
\qquad
\begin{array}{c c}
 & \begin{array}{ccc} 1 & 2 & 3 \end{array} \\
\begin{array}{l} \text{1 taxi} \\ \text{2 taxis} \\ \text{3 taxis} \end{array} & N = \begin{pmatrix} 1 & 3/2 & 5/3 \\ 1/2 & 19/20 & 4/5 \\ 1/3 & 1/2 & 17/20 \end{pmatrix}
\end{array}
$$

Outcome of first firm choosing 1 and second firm choosing 3 taxis

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

6.3 SOLVING WITH PYTHON

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

Python input

```python
import nashpy as nash


def get_game(profits):
    """Return the game object.

    Args:
        profits: a matrix with expected profits

    Returns:
        A nashpy game object
    """
    return nash.Game(profits, profits.T)
```

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

Python input

```python
import numpy as np

profits = np.array(
    (
        (1, 1 / 2, 1 / 3),
        (3 / 2, 19 / 20, 1 / 2),
        (5 / 3, 4 / 5, 17 / 20),
    )
)
game = get_game(profits=profits)
print(game)
```

which gives:

Python output

```
Bi matrix game with payoff matrices:

Row player:
[[1.         0.5        0.33333333]
 [1.5        0.95       0.5       ]
 [1.66666667 0.8        0.85      ]]

Column player:
[[1.         1.5        1.66666667]
 [0.5        0.95       0.8       ]
 [0.33333333 0.5        0.85      ]]
```

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

Python input

```python
for eq in game.support_enumeration():
    print(eq)
```

which gives:

Python output

```
(array([0., 1., 0.]), array([0., 1., 0.]))
(array([0., 0., 1.]), array([0., 0., 1.]))
(array([0. , 0.7, 0.3]), array([0. , 0.7, 0.3]))
```

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

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

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

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

Python input

```python
def get_game(profits, offset):
    """Return the game object with a given offset when 3 taxis
    are provided.

    Args:
        profits: a matrix with expected profits
        offset: a float

    Returns:
        A nashpy game object
    """
    new_profits = np.array(profits)
    new_profits[2] += offset
    return nash.Game(new_profits, new_profits.T)
```

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

Python input

```python
def get_equilibria(profits, offset):
    """Return the equilibria for a given offset when 3 taxis
    are provided.

    Args:
        profits: a matrix with expected profits
        offset: a float

    Returns:
        A nashpy game object
    """
    game = get_game(profits=profits, offset=offset)
    return tuple(game.support_enumeration())
```

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

Python input

```python
offset = 0
while len(get_equilibria(profits=profits, offset=offset)) > 1:
    offset += 0.01
```

This gives a final offset value of:

Python input

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

Python output

```
0.15
```

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

Python input

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

giving:

Python output

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

6.4 SOLVING WITH R

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

R input

```
library(Recon)

#' Returns the equilibria in pure strategies
#'
#' @param profits: a matrix with expected profits
#'
#' @return a list of equilibria
get_equilibria <- function(profits){
    sim_nasheq(profits, t(profits))
}
```

Using this we can obtain the pure Nash equilibria:

R input

```
profits <- rbind(
        c(1, 1 / 2, 1 / 3),
        c(3 / 2, 19 / 20, 1 / 2),
        c(5 / 3, 4 / 5, 17 / 20)
    )
eqs <- get_equilibria(profits = profits)
print(eqs)
```

which gives:

R output

```
$`Equilibrium 1`
[1] "2" "2"

$`Equilibrium 2`
[1] "3" "3"
```

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

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

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

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

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

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

R input

```r
#' Returns the equilibria in pure strategies
#' for a given offset
#'
#' @param profits: a matrix with expected profits
#' @param offset: a float
#'
#' @return a list of equilibria
get_equilibria <- function(profits, offset){
    new_profits <- rbind(
                profits[c(1, 2), ],
                profits[3, ] + offset)
    sim_nasheq(new_profits, t(new_profits))
}
```

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

R input

```r
offset <- 0
while (length(
            get_equilibria(profits = profits, offset = offset)
            ) > 1){
    offset <- offset + 0.01
}
```

This gives a final offset value of:

R input

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

R output

```
[1] 0.15
```

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

R input

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

giving:

R output

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

6.5 RESEARCH

TBA

CHAPTER 7

Agent Based Simulation

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

7.1 PROBLEM

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

7.2 THEORY

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

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

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

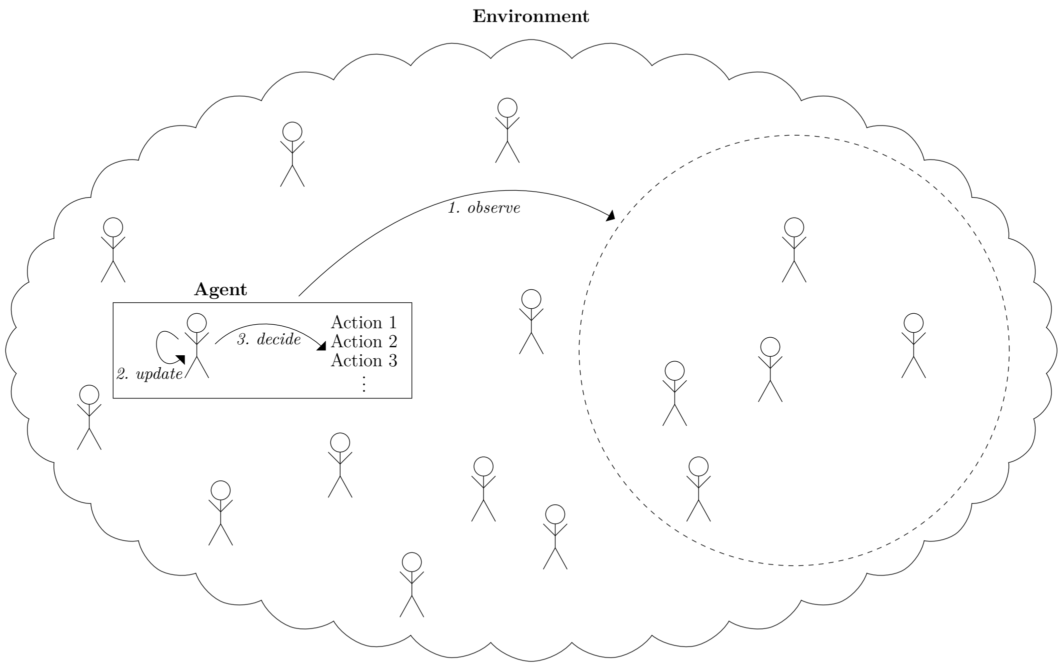


Figure 7.1 Representation of an agent interacting with its environment.

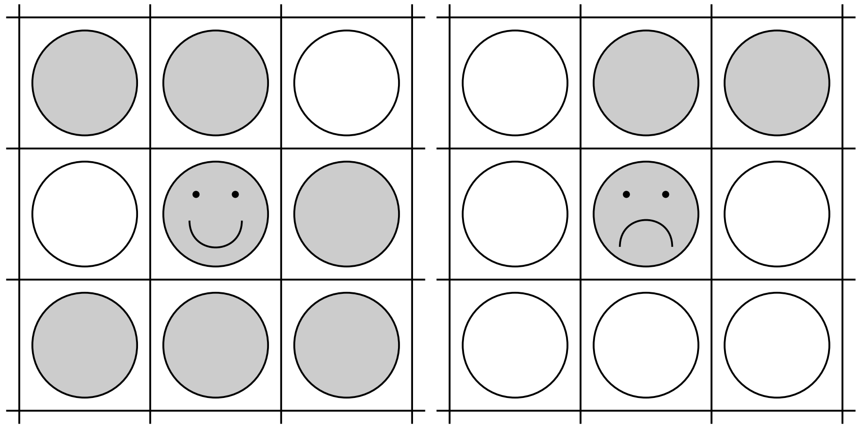
connections may change over time, possibly due to the actions taken by the agents.

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

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

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

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



(a) A happy household, with 6 similar neighbours ($\frac{6}{8} > p = 0.5$)

(b) An unhappy household, with 2 similar neighbours ($\frac{2}{8} < p = 0.5$)

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

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

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

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

Therefore our model logic is:

1. Initialise the model: fill each house in the grid with either a household of Cardiff City FC or Swansea City AFC supporters with probability 0.5 each.

2. At each discrete time step, for every house:

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

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

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

7.3 SOLVING WITH PYTHON

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

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

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

First let's import some useful libraries:

Python input

```python
import random
import itertools
import numpy as np
```

Now let's define the `City`:

Python input

```python
class City:
    def __init__(self, size, threshold):
        """Initialises the City object.

        Args:
            size: an integer number of rows and columns
            threshold: a number between 0 and 1 representing
              the minimum acceptable proportion of similar
              neighbours
        """
        self.size = size
        sides = range(size)
        self.coords = itertools.product(sides, sides)
        self.houses = {
            (x, y): House(x, y, threshold, self)
            for x, y in self.coords
        }

    def run(self, n_steps):
        """Runs the simulation of a number of time steps.

        Args:
            n_steps: an integer number of steps
        """
        for turn in range(n_steps):
            self.take_turn()

    def take_turn(self):
        """Swaps all sad households."""
        sad = [h for h in self.houses.values() if h.sad()]
        random.shuffle(sad)
        i = 0
        while i <= len(sad) / 2:
            sad[i].swap(sad[-i])
            i += 1

    def mean_satisfaction(self):
        """Finds the average household satisfaction.

        Returns:
            The average city's household satisfaction
        """
        return np.mean(
            [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 our model we only need one instance of the `City` class, however it is useful to be able to produce more in order to run multiple trials with different random seeds. This class contains four methods: `__init__`, `run`, `take_turn` and `mean_satisfaction`.

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

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

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

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

Python input

```
class House:
    def __init__(self, x, y, threshold, city):
        """Initialises the House object.

        Args:
            x: the integer x-coordinate
            y: the integer y-coordinate
            threshold: a number between 0 and 1 representing
              the minimum acceptable proportion of similar
              neighbours
            city: an instance of the City class
        """
        self.x = x
        self.y = y
        self.threshold = threshold
        self.kind = random.choice(["Cardiff", "Swansea"])
        self.city = city

    def satisfaction(self):
        """Determines the household's satisfaction level.

        Returns:
            A proportion
        """
        same = 0
        for x, y in itertools.product([-1, 0, 1], [-1, 0, 1]):
            ax = (self.x + x) % self.city.size
            ay = (self.y + y) % self.city.size
            same += self.city.houses[ax, ay].kind == self.kind
        return (same - 1) / 8

    def sad(self):
        """Determines if the household is sad.

        Returns:
            a Boolean
        """
        return self.satisfaction() < self.threshold

    def swap(self, house):
        """Swaps two households.

        Args:
            house: the house object to swap household with
        """
        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 though each of the house's neighbouring cells in the city grid, counts the number of neighbours that are of the same kind as itself, and returns this as a proportion. Then the `sad` method returns a boolean indicating of the household's satisfaction is below the minimum threshold.

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

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

Python input

```python
def find_mean_happiness(seed, size, threshold, n_steps):
    """Create and run an instance of the simulation.

    Args:
        seed: the random seed to use
        size: an integer number of rows and columns
        threshold: a number between 0 and 1 representing
            the minimum acceptable proportion of similar
            neighbours
        n_steps: an integer number of steps

    Returns:
        The average city's household satisfaction after
        n_steps
    """
    random.seed(seed)
    C = City(size, threshold)
    C.run(n_steps)
    return C.mean_satisfaction()
```

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

Python input

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

Python output

```
0.4998
```

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

Python input

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

Python output

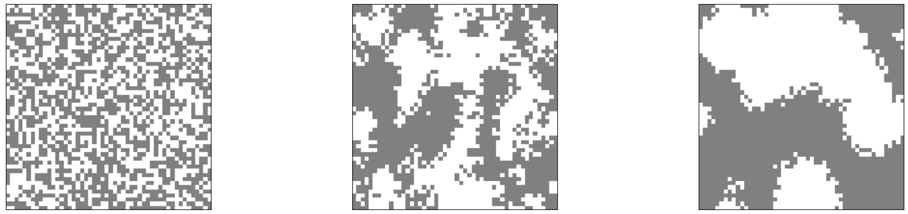
```
0.9078
```

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

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

7.4 SOLVING WITH R

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



(a) At the beginning. (b) After 20 time steps. (c) After 100 time steps.

Figure 7.3 Plotted results from the Python code.

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

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

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

Now let's define the `City`:

R input

```
library(R6)
city <- R6Class("City", list(
  size = NA,
  houses = NA,
  initialize = function(size, threshold) {
    self$size <- size
    self$houses <- c()
    for (x in 1:size) {
      row <- c()
      for (y in 1:size) {
        row <- c(row, house$new(x, y, threshold, self))
      }
      self$houses <- rbind(self$houses, row)
    } },
  run = function(n_steps) {
    if (n_steps > 0) {
      for (turn in 1:n_steps) {
        self$take_turn()
    } } },
  take_turn = function() {
    sad <- c()
    for (house in self$houses) {
      if (house$sad()) {
        sad <- c(sad, house)
      } }
    sad <- sample(sad)
    num_sad <- length(sad)
    i <- 1
    while (i <= num_sad / 2) {
      sad[[i]]$swap(sad[[num_sad - i]])
      i <- i + 1
    } },
  mean_satisfaction = function() {
    mean(sapply(self$houses, function(x) x$satisfaction()))
  })
)
```

This defines an R6 class, a template or a set of instructions that can be used to create instances of it, called objects. For our model we only need one instance of the City class, although it may be useful to be able to produce more in order to run multiple trials with different random seeds. This class contains four methods: `initialize`, `run`, `take_turn` and `mean_satisfaction`.

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

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

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

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

R input

```
house <- R6Class("House", list(
  x = NA,
  y = NA,
  threshold = NA,
  city = NA,
  kind = NA,
  initialize = function(x = NA,
                        y = NA,
                        threshold = NA,
                        city = NA) {
    self$x <- x
    self$y <- y
    self$threshold <- threshold
    self$city <- city
    self$kind <- sample(c("Cardiff", "Swansea"), 1)
  },
  satisfaction = function() {
    same <- 0
    for (x in -1:1) {
      for (y in -1:1) {
        ax <- ( (self$x + x - 1) %% self$city$size) + 1
        ay <- ( (self$y + y - 1) %% self$city$size) + 1
        if (self$city$houses[[ax, ay]]$kind == self$kind) {
          same <- same + 1
        } } }
    (same - 1) / 8
  },
  sad = function() {
    self$satisfaction() < self$threshold
  },
  swap = function(house) {
    old <- self$kind
    self$kind <- house$kind
    house$kind <- old
  })
)
```

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

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

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

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

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

R input

```r
#' Create and run an instance of the simulation.
#'
#' @param seed: the random seed to use
#' @param size: an integer number of rows and columns
#' @param threshold: a number between 0 and 1 representing
#'   the minimum acceptable proportion of similar neighbours
#' @param n_steps: an integer number of steps
#'
#' @return The average city's household satisfaction
#'   after n_steps
find_mean_happiness <- function(seed, size,
                                threshold, n_steps){
  set.seed(seed)
  our_city <- city$new(size, threshold)
  our_city$run(n_steps)
  our_city$mean_satisfaction()
}
```

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

R input

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

R output

```
[1] 0.4956
```

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

R input

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

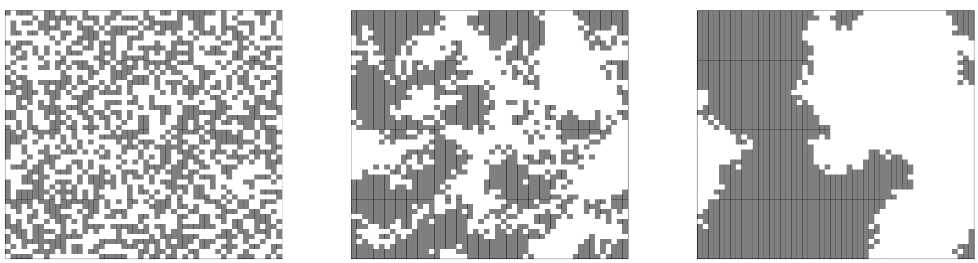
R output

```
[1] 0.9338
```

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

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

7.5 RESEARCH



(a) At the beginning. (b) After 20 time steps. (c) After 100 time steps.

Figure 7.4 Plotted results from the R code.

V

Optimisation

Linear Programming

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

8.1 PROBLEM

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

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

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

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

Table 8.1 List of modules on offer at the university.

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

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

8.2 THEORY

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

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

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

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

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

$$\text{Maximise: } 50A + 60B \tag{8.1}$$

$$\text{Subject to:}$$

$$4A + 6B \leq 24 \tag{8.2}$$

$$5A + 4B \leq 20 \tag{8.3}$$

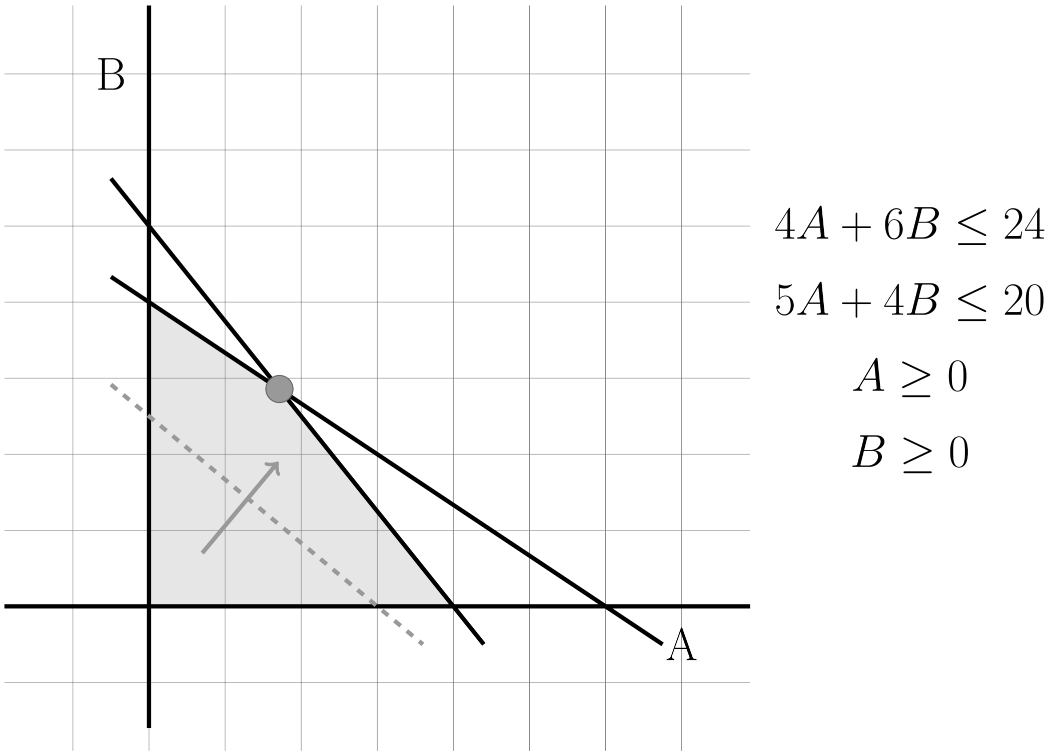


Figure 8.1 Visual representation of the paint linear program. The feasible convex region is shaded in grey; the objective function with arbitrary value is shown in a dashed line.

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

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

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

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

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

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

Additionally there are further clashes between these sets:

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

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

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

$$\text{Minimise: } \sum_{t\in T} Y_j \tag{8.4}$$

Subject to:

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

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

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

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

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

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

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

$$\text{Minimise: } c^T Z \tag{8.11}$$

Subject to:

$$AZ \star b \tag{8.12}$$

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

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

$$Z_{|M|t+m} = X_{mt} \tag{8.13}$$

$$Z_{|M|^2+m} = Y_m \tag{8.14}$$

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

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

8.3 SOLVING WITH PYTHON

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

Python input

```
Ac = [0, 1]
Ao = [2, 3, 4]
Bc = [5, 6]
Bo = [7, 8]
Cc = [9, 10]
Co = [11, 12, 13]
modules = Ac + Ao + Bc + Bo + Cc + Co
times = range(14)
```

Now let's begin by defining an empty problem:

Python input

```
import pulp

prob = pulp.LpProblem("ExamScheduling", pulp.LpMinimize)
```

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

Python input

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

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

Python input

```
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`. Let's look at $X_{2,5}$, the variable corresponding to module 2 being scheduled on day 5:

Python input

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

Python output

```
X_2_5
```

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

For the objective function, Equation 8.4:

Python input

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

Now the constraints, Inequalities 8.5-8.10:

Python input

```
M = 1 / len(modules)
for day in times:
    prob += M * sum(x[m][day] for m in modules) <= y[day]
    prob += sum([x[mod][day] for mod in Ac + Ao]) <= 1
    prob += sum([x[mod][day] for mod in Bc + Bo + Co]) <= 1
    prob += sum([x[mod][day] for mod in Bc + Bo + Ac]) <= 1
    prob += sum([x[mod][day] for mod in Cc + Co + Bo]) <= 1

for mod in modules:
    prob += sum(x[mod][day] for day in times) == 1
```

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

Now we can go ahead and solve the problem:

Python input

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

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

Python input

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

Python output

```
0.0
```

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

Python input

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

Python output

```
1.0
```

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

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

Python input

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

giving:

Python output

```
Day 0: 1, 12,
Day 5: 0, 13,
Day 6: 11,
Day 7: 4, 6, 10,
Day 8: 3, 5, 9,
Day 9: 2, 7,
Day 13: 8,
```

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

8.4 SOLVING WITH R

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

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

First we consider the objective function:

R input

```r
#' Writes the row of coefficients for the objective function
#'
#' @param n_modules: the number of modules to schedule
#' @param n_days: the maximum number of days to schedule
#'
#' @return the objective function row to minimise
write_objective <- function(n_modules, n_days){
  all_days <- rep(0, n_modules * n_days)
  Ys <- rep(1, n_days)
  append(all_days, Ys)
}
```

For 3 modules and 3 days:

R input

```r
write_objective(3, 3)
```

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

R output

```
[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

```
#' Writes the constraint row dealing with clashes
#'
#' @param clashes: a vector of module indices that all cannot
#'                 be scheduled at the same time
#' @param day: an integer representing the day
#'
#' @return the constraint row corresponding to that set of
#'         clashes on that day
write_X_clashes <- function(clashes, day, n_days, n_modules){
  today <- rep(0, n_modules)
  today[clashes] = 1
  before_today <- rep(0, n_modules * (day - 1))
  after_today <- rep(0, n_modules * (n_days - day))
  all_days <- c(before_today, today, after_today)
  full_coeffs <- c(all_days, rep(0, n_days))
  full_coeffs
}
```

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

```
#' Writes the constraint row to ensure that every module is
#' scheduled once and only one
#'
#' @param module: an integer representing the module
#'
#' @return the constraint row corresponding to scheduling a
#'         module on only one day
write_X_requirements <- function(module, n_days, n_modules){
  today <- rep(0, n_modules)
  today[module] = 1
  all_days <- rep(today, n_days)
  full_coeffs <- c(all_days, rep(0, n_days))
  full_coeffs
}
```

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

R input

```
#' Writes the constraint row representing the Y variable,
#' whether at least one exam is scheduled on that day
#'
#' @param day: an integer representing the day
#'
#' @return the constraint row corresponding to creating Y
write_Y_constraints <- function(day, n_days, n_modules){
  today <- rep(1, n_modules)
  before_today <- rep(0, n_modules * (day - 1))
  after_today <- rep(0, n_modules * (n_days - day))
  all_days <- c(before_today, today, after_today)
  all_Ys <- rep(0, n_days)
  all_Ys[day] = -n_modules
  full_coeffs <- append(all_days, all_Ys)
  full_coeffs
}
```

Finally the following function uses them all to assemble a coefficients matrix. It loops though the parameters for each constraint row required, uses the appropriate function to create the row of the coefficients matrix, sets the appropriate inequality direction ($\leq$, $=$, $\geq$), and the value of the right hand side. It returns all three components:

R input

```r
#' Writes all the constraints as a matrix, column of
#' inequalities, and right hand side column.
#'
#' @param list_clashes: a list of vectors with sets of modules
#          that cannot be scheduled at the same time
#'
#' @return f.con the LHS of the constraints as a matrix
#' @return f.dir the directions of the inequalities
#' @return f.rhs the values of the RHS of the inequalities
write_constraints <- function(list_clashes, n_days, n_modules){
  all_rows <- c()
  all_dirs <- c()
  all_rhss <- c()
  n_rows <- 0

  for (clash in list_clashes){
    for (day in 1:n_days){
      clashes <- write_X_clashes(clash, day, n_days, n_modules)
      all_rows <- append(all_rows, clashes)
      all_dirs <- append(all_dirs, "<=")
      all_rhss <- append(all_rhss, 1)
      n_rows <- n_rows + 1
    }
  }

  for (module in 1:n_modules){
    reqs <- write_X_requirements(module, n_days, n_modules)
    all_rows <- append(all_rows, reqs)
    all_dirs <- append(all_dirs, "==")
    all_rhss <- append(all_rhss, 1)
    n_rows <- n_rows + 1
  }

  for (day in 1:n_days){
    Yconstraints <- write_Y_constraints(day, n_days, n_modules)
    all_rows <- append(all_rows, Yconstraints)
    all_dirs <- append(all_dirs, "<=")
    all_rhss <- append(all_rhss, 0)
    n_rows <- n_rows + 1
  }

  f.con <- matrix(all_rows, nrow = n_rows, byrow = TRUE)
  f.dir <- all_dirs
  f.rhs <- all_rhss
  list(f.con, f.dir, f.rhs)
}
```

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

R input

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

This would give three components:

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

R output

```
[[1]]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    0    0    0    0
[2,]    0    0    1    1    0    0
[3,]    1    0    1    0    0    0
[4,]    0    1    0    1    0    0
[5,]    1    1    0    0   -2    0
[6,]    0    0    1    1    0   -2

[[2]]
[1] "<=" "<=" "==" "==" "<=" "<="

[[3]]
[1] 1 1 1 1 0 0
```

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

R input

```r
n_modules = 14
n_days = 14

Ac <- c(0, 1)
Ao <- c(2, 3, 4)
Bc <- c(5, 6)
Bo <- c(7, 8)
Cc <- c(9, 10)
Co <- c(11, 12, 13)

list_clashes <- list(
  c(Ac, Ao),
  c(Bc, Bo, Co),
  c(Bc, Bo, Ac),
  c(Bo, Cc, Co)
)
```

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

R input

```r
constraints <- write_constraints(list_clashes = list_clashes,
                                 n_days = n_days,
                                 n_modules = n_modules)
f.con <- constraints[[1]]
f.dir <- constraints[[2]]
f.rhs <- constraints[[3]]
f.obj <- write_objective(n_modules = n_modules, n_days = n_days)
```

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

R input

```r
library(ROI)

milp <- OP(objective = L_objective(f.obj),
           constraints = L_constraint(L = f.con,
                                      dir = f.dir,
                                      rhs = f.rhs),
           types = rep("B", length(f.obj)),
           maximum = FALSE)
```

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

Now to solve:

R input

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

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

R input

```r
print(sol$solution)
```

R output

```
  [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
 [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
 [59] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
 [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
[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
[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
[175] 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 1 0
[204] 1 0 1 1 1 0 1
```

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

R input

```r
for (day in 1:n_days){
  if (sol$solution[(n_days * n_modules) + day] == 1){
    schedule <- paste("Day", day, ":")
    for (module in 1:n_modules){
      var <- ((day - 1) * n_modules) + module
      if (sol$solution[var] == 1){
        schedule <- paste(schedule, module)
      }
    }
    print(schedule)
  }
}
```

R output

```
[1] "Day 2 : 4 11"
[1] "Day 6 : 1 12"
[1] "Day 8 : 7"
[1] "Day 10 : 8"
[1] "Day 11 : 3 13"
[1] "Day 12 : 2 6 9 14"
[1] "Day 14 : 5 10"
```

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

8.5 RESEARCH

CHAPTER 9

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 we finding an almost perfect choice quickly is just as good. This is where the field of heuristics comes in to play.

9.1 PROBLEM

Consider a delivery company that needs to find itineraries for a driver. In the past, the management team has noticed that drivers will often drive to whichever next stop is closest but this often makes for longer deliveries.

The stops are represented in Figure 9.2.

The distance matrix is given in equation (9.1).

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

The value d gives the travel distance between stops i and j. For example, $d_{23} = 89$ indicates that the distance between the 2nd and 3rd stop in the third itinerary is given 89.

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

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

9.2 THEORY

The heuristic approach take here will be to use a neighborhood search algorithm. This algorithm works by considering a given potential solution, evaluating it and then trying another potential solution *close* to it. What *close* means depends on

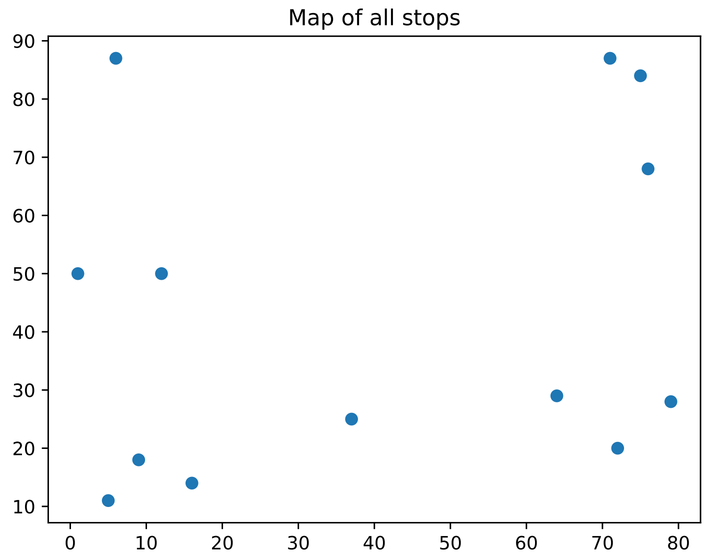


Figure 9.1 Diagrammatic representation of the action sets and payoff matrices for the game.

different approaches and problems: it is referred to as the neighbourhood. As a new solution is evaluated if it is *good* (this is again a term that depends on the approach and problem) then the search continues from the neighbourhood of this new solution.

For our problem, the first aspect of this is to represent a given trajectory between all the potential stops as a *tour*. If we have 3 total stops and require that the tour starts and stops at the first one then there are two possible tours:

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

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

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

Thus, with:

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

We have:

$$c((1,2,3,1)) = d_{12} + d_{23} + d_{31} = 1 + 15 + 3 = 19$$
$$c((1,3,2,1)) = d_{13} + d_{32} + d_{21} = 3 + 3 + 1 = 7$$

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

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

This is shown diagrammatically in Figure 9.2.

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

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

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

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

One possible neighbourhood is to choose 2 stops at random and swap. For example, the tour $t^{(1)} \in N(t)$ is obtained by swapping the 3rd and 5th stops.

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

Another possible neighbourhood is to choose 2 stops at random and reversing the order of all stops between (including) those two stops. For example, the tour $t^{(2)} \in N(t)$ is obtained by reversing the order of all stops between the 3rd and the 5th stop.

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

Examples of these tours are shown in Figure 9.3.

9.3 SOLVING WITH PYTHON

To solve this problem using Python we will write functionality that matches the first three steps in the Section 9.2.

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

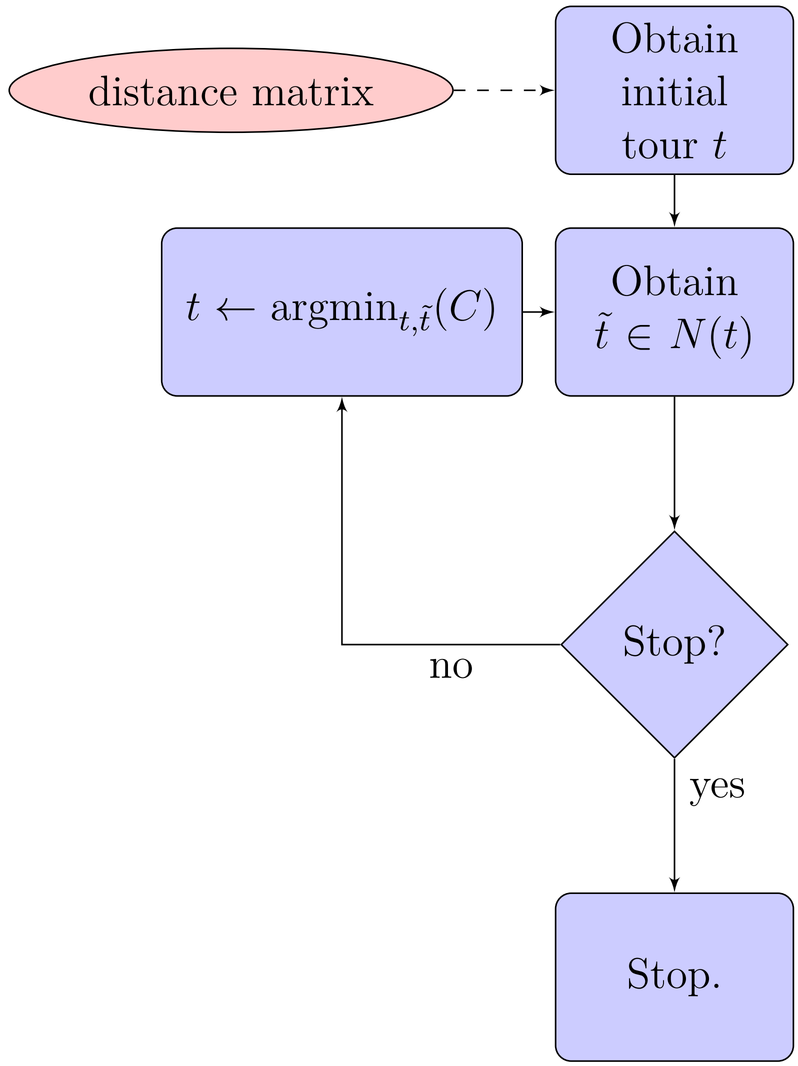


Figure 9.2 The general neighbourhood search algorithm. $N(t)$ refers to some neighbourhood of t.

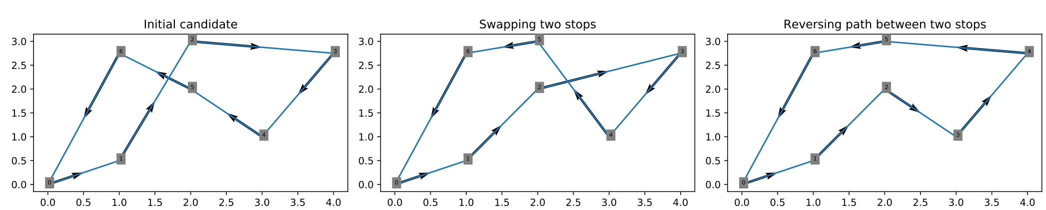


Figure 9.3 The effect of two neighborhood operators on t. $t^{(1)}$ is obtained by swapping stops 3 and 5. $t^{(2)}$ is obtained by reversing the path between stops 3 and 5.

Python input

```python
import numpy as np


def get_initial_candidate(number_of_stops, seed=None):
    """Return an initial tour.

    Args:
        number_of_stops: The number of stops
        seed: An integer seed. If an integer value is
              passed it will create a random tour.

    Returns:
        A tour starting an ending at stop with index 0.
    """
    internal_stops = list(range(1, number_of_stops))
    if seed is not None:
        np.random.seed(seed)
        np.random.shuffle(internal_stops)
    return [0] + internal_stops + [0]
```

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

Python input

```python
number_of_stops = 13
seed = 0
initial_candidate = get_initial_candidate(
    number_of_stops=number_of_stops,
    seed=seed,
)
print(initial_candidate)
```

Python output

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

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

Python input

```python
def get_cost(tour, distance_matrix):
    """Return the cost of a tour.

    Args:
        tour: A given tuple of successive stops.
        distance_matrix: The distance matrix of the problem.

    Returns:
        The cost
    """
    return sum(
        distance_matrix[current_stop, next_stop]
        for current_stop, next_stop in zip(tour[:-1], tour[1:])
    )
```

Python input

```
distance_matrix = np.array(
    (
        (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),
    )
)
cost = get_cost(
    tour=initial_candidate,
    distance_matrix=distance_matrix,
)
print(cost)
```

Python output

```
827
```

We will now define two different neighbourhood operators:

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

Python input

```python
def swap_stops(tour):
    """Return a new tour by swapping two stops.

    Args:
        tour: A given tuple of successive stops.

    Returns:
        A tour
    """
    number_of_stops = len(tour) - 1
    i, j = sorted(
        np.random.choice(range(1, number_of_stops), 2)
    )
    new_tour = list(tour)
    new_tour[i], new_tour[j] = tour[j], tour[i]
    return new_tour


def reverse_path(tour):
    """Return a new tour by reversing the path between two
    stops.

    Args:
        tour: A given tuple of successive stops.

    Returns:
        A tour
    """
    number_of_stops = len(tour) - 1
    i, j = sorted(
        np.random.choice(range(1, number_of_stops), 2)
    )
    new_tour = tour[:i] + tour[i : j + 1][::-1] + tour[j + 1 :]
    return new_tour
```

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

Python input

```python
print(swap_stops(initial_candidate))
```

which swaps the 3rd and 8th stops:

Python output

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

Python input

```python
print(reverse_path(initial_candidate))
```

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

Python output

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

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

Python input

```python
def run_neighbourhood_search(
    distance_matrix,
    number_of_stops,
    iterations,
    seed=None,
    neighbourhood_operator=reverse_path,
):
    """Returns a tour by carrying out a neighbourhood search.

    Args:
        distance_matrix: the distance matrix
        number_of_stops: the number of stops
        iterations: the number of iterations for which to
                    run the algorithm
        seed: a random seed (default: None)
        neighbourhood_operator: the neighbourhood operator
                                (default: reverse_path)

    Returns:
        A tour
    """
    candidate = get_initial_candidate(
        number_of_stops=number_of_stops,
        seed=seed,
    )

    best_cost = get_cost(
        tour=candidate,
        distance_matrix=distance_matrix,
    )

    for _ in range(iterations):
        new_candidate = neighbourhood_operator(candidate)
        if (
            cost := get_cost(
                tour=new_candidate,
                distance_matrix=distance_matrix,
            )
        ) <= best_cost:
            best_cost = cost
            candidate = new_candidate

    return candidate
```

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

Python input

```
number_of_iterations = 1000

solution_with_swap_stops = run_neighbourhood_search(
    distance_matrix=distance_matrix,
    number_of_stops=number_of_stops,
    iterations=number_of_iterations,
    seed=seed,
    neighbourhood_operator=swap_stops,
)
print(solution_with_swap_stops)
```

giving:

Python output

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

Python input

```
solution_with_reverse_path = run_neighbourhood_search(
    distance_matrix=distance_matrix,
    number_of_stops=number_of_stops,
    iterations=number_of_iterations,
    seed=seed,
    neighbourhood_operator=reverse_path,
)
print(solution_with_reverse_path)
```

giving:

Python output

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

Importantly, the costs differ substantially:

Python input

```python
cost = get_cost(
    tour=solution_with_swap_stops,
    distance_matrix=distance_matrix,
)
print(cost)
```

which gives:

Python output

```
362
```

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

Python input

```python
cost = get_cost(
    tour=solution_with_reverse_path,
    distance_matrix=distance_matrix,
)
print(cost)
```

which gives:

Python output

```
299
```

9.4 SOLVING WITH R

To solve this problem using R we will write functionality that matches the first three steps in the Section 9.2.

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

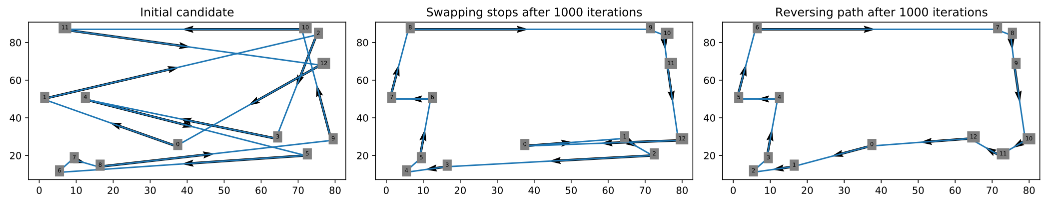


Figure 9.4 The final tours obtained by using the neighbourhood search in Python.

R input

```r
#' Return an initial tour.
#'
#' @param number_of_stops The number of stops.
#' @param seed An integer seed. If an integer value is
#'        passed it will create a random tour.
#'
#' @return A tour starting an ending at stop with index 0.
get_initial_candidate <- function(number_of_stops, seed = NA){
    internal_stops <- 1:(number_of_stops - 1)
    if (!is.na(seed)) {
      set.seed(seed)
      internal_stops <- sample(internal_stops)
    }
    c(0, internal_stops, 0)
}
```

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

R input

```r
number_of_stops <- 13
seed <- 0
initial_candidate <- get_initial_candidate(
    number_of_stops = number_of_stops,
    seed = seed)
print(initial_candidate)
```

R output

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

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

R input

```r
#' Return the cost of a tour
#'
#' @param tour A given vector of successive stops.
#' @param seed The distance matrix of the problem.
#'
#' @return The cost
get_cost <- function(tour, distance_matrix){
    pairs <-  cbind(tour[-length(tour)], tour[-1]) + 1
    sum(distance_matrix[pairs])
}
```

R input

```
distance_matrix <- rbind(
        c(0, 35, 35, 29, 70, 35, 42, 27, 24, 44, 58, 71, 69),
        c(35, 0, 67, 32, 72, 40, 71, 56, 36, 11, 66, 70, 37),
        c(35, 67, 0, 63, 64, 68, 11, 12, 56, 77, 48, 67, 94),
        c(29, 32, 63, 0, 93, 8, 71, 56, 8, 33, 84, 93, 69),
        c(70, 72, 64, 93, 0, 101, 56, 56, 92, 81, 16, 5, 69),
        c(35, 40, 68, 8, 101, 0, 76, 62, 11, 39, 91, 101, 76),
        c(42, 71, 11, 71, 56, 76, 0, 15, 65, 81, 40, 60, 94),
        c(27, 56, 12, 56, 56, 62, 15, 0, 50, 66, 41, 58, 82),
        c(24, 36, 56, 8, 92, 11, 65, 50, 0, 39, 81, 91, 74),
        c(44, 11, 77, 33, 81, 39, 81, 66, 39, 0, 77, 79, 37),
        c(58, 66, 48, 84, 16, 91, 40, 41, 81, 77, 0, 20, 73),
        c(71, 70, 67, 93, 5, 101, 60, 58, 91, 79, 20, 0, 65),
        c(69, 37, 94, 69, 69, 76, 94, 82, 74, 37, 73, 65, 0)
)
cost <- get_cost(
    tour = initial_candidate,
    distance_matrix = distance_matrix)
print(cost)
```

R output

```
[1] 709
```

We will now define two different neighbourhood operators:

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

R input

```r
#' Return a new tour by swapping two stops.
#'
#' @param tour A given vector of successive stops.
#'
#' @return A tour
swap_stops <- function(tour){
    number_of_stops <- length(tour) - 1
    stops_to_swap <- sort(sample(2:number_of_stops, 2))
    new_tour <- replace(x = tour,
                        list = stops_to_swap,
                        values = rev(tour[stops_to_swap]))
    }

#' Return a new tour by reversing the path between two stops.
#'
#' @param tour A given vector of successive stops.
#'
#' @return A tour
reverse_path <- function(tour){
    number_of_stops <- length(tour) - 1
    stops_to_swap <- sort(sample(2:number_of_stops, 2))
    i <- stops_to_swap[1]
    j <- stops_to_swap[2]
    new_order <- c(
            c(1: (i - 1)),
            c(j:i),
            c( (j + 1): length(tour))
            )
    tour[new_order]
    }
```

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

R input

```r
print(swap_stops(initial_candidate))
```

which swaps the 6th and 11th stops:

R output

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

R input

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

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

R output

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

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

R input

```r
#' Returns a tour by carrying out a neighbourhood search
#'
#' @param distance_matrix: the distance matrix
#' @param number_of_stops: the number of stops
#' @param iterations: the number of iterations for
#'                    which to run the algorithm
#' @param seed: a random seed (default: None)
#' @param neighbourhood_operator: the neighbourhood operation
#'                                (default: reverse_path)
#'
#' @return A tour
run_neighbourhood_search <- function(
  distance_matrix,
  number_of_stops,
  iterations,
  seed = NA,
  neighbourhood_operator = reverse_path
){
  candidate <- get_initial_candidate(
    number_of_stops = number_of_stops,
    seed = seed
    )

  best_cost <- get_cost(
    tour = candidate,
    distance_matrix = distance_matrix
    )

  for (repetition in 1:iterations) {
    new_candidate <- neighbourhood_operator(candidate)
    cost <- get_cost(
        tour = new_candidate,
        distance_matrix = distance_matrix)

    if (cost <= best_cost) {
      best_cost <- cost
      candidate <- new_candidate
    }

  }
  candidate
}
```

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

R input

```
number_of_iterations <- 1000
solution_with_swap_stops <- run_neighbourhood_search(
    distance_matrix = distance_matrix,
    number_of_stops = number_of_stops,
    iterations = number_of_iterations,
    seed = seed,
    neighbourhood_operator = swap_stops
)
print(solution_with_swap_stops)
```

giving:

R output

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

R input

```
number_of_iterations <- 1000
solution_with_reverse_path <- run_neighbourhood_search(
    distance_matrix = distance_matrix,
    number_of_stops = number_of_stops,
    iterations = number_of_iterations,
    seed = seed,
    neighbourhood_operator = reverse_path
)
print(solution_with_reverse_path)
```

giving:

R output

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

Importantly, the costs differ substantially:

R input

```r
cost <- get_cost(
    tour = solution_with_swap_stops,
    distance_matrix = distance_matrix
)
print(cost)
```

which gives:

R output

```
[1] 373
```

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

R input

```r
cost <- get_cost(
    tour = solution_with_reverse_path,
    distance_matrix = distance_matrix
)
print(cost)
```

which gives:

R output

```
[1] 299
```

9.5 RESEARCH

TBA

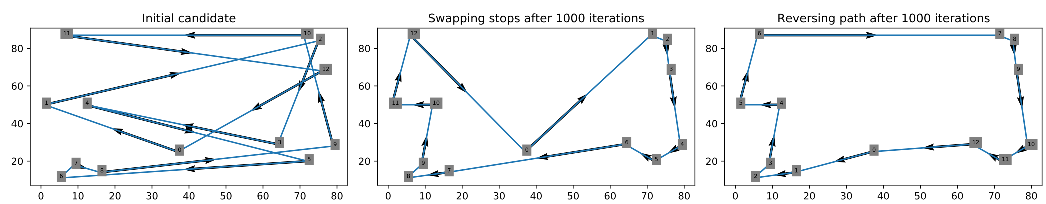


Figure 9.5 The final tours obtained by using the neighbourhood search in R

Bibliography