

---

*Half Title Page*



---

# *Title Page*

---

*LOC Page*

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



---

# Contents

---

Foreword	ix
Preface	xi
Contributors	xiii
<b>SECTION I Getting Started</b>	
<b>CHAPTER 1 ■ Introduction</b>	<b>3</b>
1.1 WHO IS THIS BOOK FOR?	3
1.2 WHAT DO WE MEAN BY APPLIED MATHEMATICS?	3
1.3 WHAT IS OPEN SOURCE SOFTWARE	4
1.4 HOW TO GET THE MOST OUT OF THIS BOOK	4
<b>CHAPTER 2 ■ Software</b>	<b>7</b>
2.1 SOFTWARE INSTALLATION	7
2.1.1 Installing Python	8
2.1.2 Installing R	8
2.2 USING THE COMMAND LINE	8
2.3 BASIC PYTHON	13
2.4 BASIC R	18
2.5 A NOTE ON HOW CODE IS DISPLAYED IN THIS BOOK	24
<b>SECTION II Probabilistic Modelling</b>	
<b>CHAPTER 3 ■ Markov Chains</b>	<b>29</b>
3.1 PROBLEM	29
3.2 THEORY	29
3.3 SOLVING WITH PYTHON	31

3.4	SOLVING WITH R	38
3.5	RESEARCH	45
CHAPTER	4 ■ Discrete Event Simulation	47
4.1	PROBLEM	47
4.2	THEORY	48
4.2.1	Event Scheduling Approach	49
4.2.2	Process Based Simulation	50
4.3	SOLVING WITH PYTHON	50
4.4	SOLVING WITH R	57
4.5	RESEARCH	63
SECTION III Dynamical Systems		
CHAPTER	5 ■ Systems dynamics	67
5.1	PROBLEM	67
5.2	THEORY	67
5.3	SOLVING WITH PYTHON	70
5.4	SOLVING WITH R	78
5.5	RESEARCH	84
Bibliography		85



---

# 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

**xiv ■ Contributors**

**Jonathan Carter**

Lawrence Berkeley National Laboratory  
Berkeley, California

**Zizhong Chen**

Jacksonville State University  
Jacksonville, Alabama

**Joseph R. Crobak**

Rutgers, The State University of New  
Jersey

Piscataway, New Jersey

**Roxana E. Diaconescu**

Yahoo! Inc.  
Burbank, California

**Roxana E. Diaconescu**

Yahoo! Inc.  
Burbank, California

# I

---

## Getting Started





# Introduction

---

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

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

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

## 1.1 WHO IS THIS BOOK FOR?

---

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

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

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

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

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

## 1.2 WHAT DO WE MEAN BY APPLIED MATHEMATICS?

---

We consider this book to be a book on applied mathematics. This is not however a universal term, for some applied mathematics is the study of mechanics and involves

modelling projectiles being fired out of canons. We will use the term a bit more freely here and mean any type of real world problem that can be tackled using mathematical tools. This is sometimes referred to as operational research, operations research, mathematical modelling or indeed just mathematics.

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

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

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

### 1.3 WHAT IS OPEN SOURCE SOFTWARE

---

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

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

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

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

### 1.4 HOW TO GET THE MOST OUT OF THIS BOOK

---

The book itself is open source. You can find the source files for this book online at [github.com/drvinceknight/ampwoss](https://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.



# Software

---

THIS book will involve using software, the particular interface to software we will use is to write code. There are numerous reasons why this is the correct way to do things but one of them is reproducibility.

This chapter will go over the basics of getting your computer set up to use the software discussed in this book: the programming languages R and Python. It will also briefly discuss using the command line: a particular interface to your whole computer. Finally it will give a brief introduction to R and Python.

This chapter (and indeed this whole book) is not a place to learn R and Python completely. We will cover specific tasks and how to carry them out in each language, but we will not cover the every intricacy of each language. There are numerous sources (books, websites, courses) that are available to do that. A lot of these places would argue that you should not learn multiple programming languages from one book, and instead concentrate on a single skill at a time. We agree, and the single skill to concentrate on with this book is the use of software to solve applied mathematical problems. The particular software itself is not the most important component.

## 2.1 SOFTWARE INSTALLATION

---

There are a number of different places from which you can buy your vegetables, you can grow them yourself, you can go to a market and pick fresh fruit from specific stalls, you can go to a supermarket and buy a bag of a collection of vegetables and in some places you can even get a box of vegetables regularly posted to you. Software is similar, there are a variety of places from which you can get it and a number of different forms in which it can be obtained.

If you're comfortable with using R and Python then you probably do not need to read this section and you might even use different so called "distributions" of each piece of software, but for the purpose of this book here is where we will be getting what we need:

- Python: we will use the Anaconda distribution: <https://www.anaconda.com/distribution/>
- R: we will be getting this directly from the Comprehensive R Archive Network (commonly referred to as CRAN): <https://cran.r-project.org>. We will also use another piece of software called Rstudio: <https://rstudio.com>.

### 2.1.1 Installing Python

Installing Python and all the software we need around it is done by downloading and running the installer for the Anaconda distribution.

1. Go to this webpage: <https://www.anaconda.com/download/>.
2. Identify and download the version of Python 3 for your operating system (Windows, Mac OSX, Linux). Run the installer.

### 2.1.2 Installing R

There are actually two pieces of software we need to install to use R for the purposes of this book, first the R language itself and second an application with which we will write R code.

1. Go to this webpage: <https://cran.r-project.org>.
2. Identify and download the latest version of R for your operating system (Windows, Mac OSX, Linux). Run the installer.
3. Go to this webpage: <https://rstudio.com>.
4. Identify and download the latest version of Rstudio for your operating system (Windows, Mac OSX, Linux). Run the installer.

## 2.2 USING THE COMMAND LINE

---

There are various interfaces to using a computer, the most common one is to use a mouse and keyboard and click on programmes we want to use. Another approach is to use what is called a command line interface this is where we do not interact graphically with a computer but we type in specific commands.

We can use our command line to navigate the various directories on our computer. There are two types of operating systems that we consider here:

- Windows
- Nix: this includes OSX (the Mac operating system) and Linux

Not all commands are the same on each type of operating system. So let us start by opening our command line interface:

- Windows: after having installed Anaconda look to open the Anaconda Prompt. There are a number of other command line interfaces available but this is the one we recommend for the purposes of this book.
- Nix: look to open the Terminal.

This should open something that looks like and somewhat resembles a black box with some text in it. This is where we will write our commands to the computer.

For example to list the contents of the directory we are currently in:

**On nix:**

Cli input

1    `ls`

**On Windows**

Cli input

2    `dir`

It is also possible to get the name of the directory we are currently in:

**On nix:**

Cli input

3    `pwd`

**On Windows**

Cli input

4    `cd`

Finally we can also use the command line to move to another directory. The command for this are the same on Nix and on Windows.

Cli input

5    `cd <name_of_subdirectory>`

The command line is an important tool to learn to use when doing tasks:

- If we want to scale the tasks, a commonly heard phrase is that ‘mouse clicks do not scale’ highlighting that to repeat a task many times when using a graphical interface is inefficient.

## 10 ■ Applied mathematics problems with Open Source Software: Operational Research with Python and R.

- If we want someone else to be able to repeat the tasks, we can use screenshots of graphical interfaces but there will always be a level of ambiguity whereas the commands used in the command line are precise.

We can use our two programming languages right within the command line interface (we will actually be using a different tool that we will describe shortly).

To use Python, simply type the following and press Enter:

Cli input

```
python
```

This should make something like the following appear:

Cli output

```
Python 3.7.1 | packaged by conda-forge | (default, Nov 13 2018, 10:30:07)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is a prompt ready to accept a Python command. Let us start with the following:

Python input

```
>>> 2 + 2
```

When you press Enter, this will give:

Python output

```
4
```

This particular way of using Python is called a REPL which stands for: ‘Read Eval Print Loop’ which indicates that it takes a command, evaluates it and waits for the next one.

To quit Python’s REPL type the following (note that `()`, more about that later):



## Python input

13 `>>> quit()`

We can do the same for R. To start R's REPL, in your command line type the following and press Enter:

## Cli input

14 `R`

This should make something like the following appear:

## Cli output

```

15 R version 3.5.1 (2018-07-02) -- "Feather Spray"
16 Copyright (C) 2018 The R Foundation for Statistical Computing
17 Platform: x86_64-apple-darwin13.4.0 (64-bit)
18
19 R is free software and comes with ABSOLUTELY NO WARRANTY.
20 You are welcome to redistribute it under certain conditions.
21 Type 'license()' or 'licence()' for distribution details.
22
23   Natural language support but running in an English locale
24
25 R is a collaborative project with many contributors.
26 Type 'contributors()' for more information and
27 'citation()' on how to cite R or R packages in publications.
28
29 Type 'demo()' for some demos, 'help()' for on-line help, or
30 'help.start()' for an HTML browser interface to help.
31 Type 'q()' to quit R.
32
33 >

```

The `>` is a prompt ready to accept an R command. Let us start with the following:

R input

```
34 > 2 + 2
```

When you press Enter, this will give:

R output

```
35 4
```

To quit R's REPL type the following:

R input

```
36 > q()
```

This will bring up a further prompt asking you to save some information about what you just did. You can type `n` for now:

R input

```
37 > Save workspace image? [y/n/c]: n
```

These two REPLs are not unique and also not the most efficient way of using the languages, however they can at times be useful if you just want to type a very short command or perhaps check something quickly.

Another approach is to save a collection of commands in a plain text file and pass it to the interpreter at the command line.

For example, if we had a number of Python commands in `main.py` we could run this at the command line using:

Cli input

```
38 python main.py
```

Similarly for a file with a number of R commands `main.R`:

Cli input

39

```
Rscript main.R
```

These are just a few of many ways to use Python and R. An important notion to understand is that Python and R are not the particular tools that we use to interface to them. On a day to day basis the authors of this book will use both of the above approaches as well as the next ones, we recommend readers take time to experiment and understand the particular use cases for which each tool works best for them.

The two tools we recommend to use in this book are:

- For Python: the Jupyter notebook, a tool that behaves similarly to a REPL, runs in the web browser and is very popular in research.
- For R: RStudio, an integrated development environment with a lot of helpful features.

The best way to start the Jupyter notebook is to type the following in your command line:

Cli input

40

```
jupyter notebook
```

This will create a *notebook server* that runs on your computer and should open a page that looks like Note that despite running in a web browser this does not need the internet to run.

We can create a new notebook and write and run code in the *cells*.

To start Rstudio, locate the application on your computer and double click on it. This will open an application that looks like

Rstudio includes its own REPL, so we can type and run single commands there but we can also write in a file that we can run

In the next sections we will cover some basics of Python and R.

## 2.3 BASIC PYTHON

This section gives a very brief overview of some introductory aspects of Python, there are excellent resources available for learning Python and we recommend the reader goes there if they feel they need an in depth understanding of the language

In the previous section, we saw how to get Python to perform a single calculation:

Python input

```
41 print(3 + 5)
```

which will give:

Python output

```
42 8
```

We can also assign values to a variable:

Python input

```
43 a = 3
44 b = 5
45 c = a + b
46 print(c)
```

This makes a point at 3 etc...

which will give:

Python output

```
47 8
```

There are a number of different types of variables in Python, here is a very brief list of some of them:

- Integers – `int` – for example 2, 4, -459060.
- Floats – `float` – for example 2.0, 3.4, -3.459060.
- Strings – `str` – for example "two", "hello world", "3450".
- Booleans – `bool` – for example `True` or `False`.

Based on the values of a variable it is possible to construct Booleans:

## Python input

```
48 is_a_larger_than_b = a > b
```

The variable `is_a_larger_than_b` will be the boolean variable `False`.

This is an important concept as boolean variable allow us to use conditional statements that let us write code that does specific things based on the value of variables. For example the following code will add 5 to the smallest variable:

## Python input

```
49 a = 3
50 b = 5
51 if a < b:
52     a = a + 3
53 elif a > b:
54     b = b + 5
55 else:
56     a = a + 3
57     b = b + 3
58 print(a, b)
```

which gives:

## Python output

```
59 6 5
```

If you are experimenting by typing the code as you go change the value of `a` or `b` to see how the behaviour changes. What happens if they are equal?

It is also possible to use these conditional statements to repeat code. For example the following code will repeatedly add 1 to the smallest variable until it becomes equal to the largest one:

## Python input

```

60 a = 3
61 b = 5
62 while a != b:
63     if a < b:
64         a = a + 1
65     else:
66         b = b + 1

```

It is important to be able to reuse code, this is done using a programming concept called a *function*, which acts similarly to a mathematical function.

The following code, creates a function that takes two variables as input and outputs the largest number and the smallest increased by 3.

## Python input

```

67 def add_3_to_smallest(a, b):
68     """This function adds 3 to the smallest of a or b."""
69     if a < b:
70         return a + 3, b
71     return a, b + 3

```

Once we have defined the function, the following is how we use it:

## Python input

```

72 print(add_3_to_smallest(a=5, b=-42))

```

which gives:

## Python output

```

73 (5, -39)

```

Python has a type of variable that is in fact a collection of pointers to other variables. This is called a list. Here for example is a collection of strings:

## Python input

```
74 tennis_players = [  
75     "Federer",  
76     "S. Williams",  
77     "V. Williams",  
78     "King",  
79 ]
```

There are a number of things that can be done with lists but one particular aspect is that they are a sub type of something called an iterable in Python which means we can iterate over them. We do this in Python using a **for** loop. For example, the following code will iterate over the list and print all the values:

## Python input

```
80 for name in tennis_players:  
81     print(name)
```

which gives:

## Python output

```
82 Federer  
83 S. Williams  
84 V. Williams  
85 King
```

We will often want to iterate over a set of integers, Python has a **range** command that can create such a set with ease. The following code will print every 3 integers from 30 to 50:

## Python input

```
86 for integer in range(30, 50, 3):  
87     print(integer)
```

which will give:

Python output

```

88 30
89 33
90 36
91 39
92 42
93 45
94 48

```

A final important aspect of Python is that of libraries. The code examples above are from the so called ‘standard library’ but Python has numerous libraries specific to given problems. A lot of these libraries came bundled with the anaconda distribution but if you want to download one that is not you can always do so as long as you have an internet connection.

For example, to download a library for studying queueing systems `ciw` open your command line interface and type the following:

Cli input

```

95 pip install ciw

```

Once you restart your python interpreter, for example if you are using a Jupyter notebook then restart the Kernel, you can then run the following to make `ciw` available to you:

Python input

```

96 import ciw

```

## 2.4 BASIC R

This section gives a very brief overview of some introductory aspects of R, there are excellent resources available for learning R [1] and we recommend the reader goes there if they feel they need an in depth understanding of the language

In the previous section, we saw how to get R to perform a single calculation:



R input

97 

```
print(3 + 5)
```

which will give:

R output

98 

```
[1] 8
```

We can also assign values to a variable:

R input

99 

```
a <- 3  
100 b <- 5  
101 c <- a + b  
102 print(c)
```

which will give:

R output

103 

```
[1] 8
```

An important difference between R and Python is that in R the base structure is in fact a vector, even if it only contains a single variable. We can use the `c` command to *concatenate* these base structures together:

R input

104 

```
print(c(a, 4))
```

giving:

R output

```
105 [1] 3 4
```

There are a number of different types of variables in R, here is a very brief list of some of them:

- Integers – `integer` – for example 2, 4, -459060.
- Floats – `double` – for example 2.0, 3.4, -3.459060.
- Strings – `character` – for example "two", "hello world", "3450".
- Booleans – `logical` – for example `TRUE` or `FALSE`.

Based on the values of a variable it is possible to construct Booleans:

R input

```
106 is_a_larger_than_b <- a > b
```

The variable `is_a_larger_than_b` will be the boolean variable `FALSE`.

This is an important concept as boolean variable allow us to use conditional statements that let us write code that does specific things based on the value of variables. For example the following code will add 5 to the smallest variable:

R input

```
107 a <- 3
108 b <- 5
109 if (a < b) {
110   a <- a + 3
111 } else if (a > b) {
112   b <- b + 3
113 } else {
114   a <- a + 3
115   b <- b + 3
116 }
117 print(c(a, b))
```

which gives:

## R output

```
118 [1] 6 5
```

If you are experimenting by typing the code as you go, change the value of `a` or `b` to see how the behaviour changes. What happens if they are equal?

R is a so called “vectorized” language which means that there is often a more appropriate approach to doing things repeatedly using vectors. This applies to the `if` statement in that there exists a `ifelse` statement that applies to vectors of booleans. For example:

## R input

```
119 booleans <- c(FALSE, TRUE, FALSE, FALSE)
120 print(ifelse(booleans, "cat", "dog"))
```

which gives:

## R output

```
121 [1] "dog" "cat" "dog" "dog"
```

It is also possible to use conditional statements to repeat code. For example the following code will repeatedly add 1 to the smallest variable until it becomes equal to the largest one:

## R input

```
122 a <- 3
123 b <- 5
124 while (a != b) {
125   if (a < b) {
126     a <- a + 1
127   }
128   else {
129     b <- b + 1
130   }
131 }
```

It is important to be able to reuse code, this is done using a programming concept called a *function*, which acts similarly to a mathematical function.

The following code creates a function that takes two variables as input and outputs the largest number and the smallest increased by 3.

R input

```

132 add_3_to_smallest <- function(a, b) {
133   # This function adds 3 to the smallest of a or b.
134   if (a < b) {
135     return(c(a + 3, b))
136   }
137   else {
138     return(c(a, b + 3))
139   }
140 }

```

Note that R will implicitly return the last computed expression without the need for a `return` statement. So the above can also be written as:

R input

```

141 add_3_to_smallest <- function(a, b) {
142   # This function adds 3 to the smallest of a or b.
143   if (a < b) {
144     c(a + 3, b)
145   }
146   else {
147     c(a, b + 3)
148   }
149 }

```

Once we have defined the function, the following is how we use it:

R input

```

150 print(add_3_to_smallest(a = 5, b = -42))

```

which gives:

## R output

```
151 [1] 5 -39
```

It is possible to iterate over elements inside R vectors:

## R input

```
152 tennis_players <- c("Federer",  
153                     "S. Williams",  
154                     "V. Williams",  
155                     "King")
```

The following will print all the names contained in the vector:

## R input

```
156 for (name in tennis_players) {  
157   print(name)  
158 }
```

which gives:

## R output

```
159 [1] "Federer"  
160 [1] "S. Williams"  
161 [1] "V. Williams"  
162 [1] "King"
```

We will often want to iterate over a vector of integers, R has a `seq` command that can create such a vector with ease. The following code will print every 3 integers from 30 to 50:

## R input

```

163 for (i in seq(30, 50, 3)) {
164   print(i)
165 }

```

which will give:

## R output

```

166 [1] 30
167 [1] 33
168 [1] 36
169 [1] 39
170 [1] 42
171 [1] 45
172 [1] 48

```

A final important aspect of R is that of packages. The code examples above are from the so called ‘base R’ but R has numerous packages specific to given problems. If you want to download and use one you can always do so as long as you have an internet connection.

For example, to download a very common collection of data science tools called *tidyverse* we use the following line of code inside of an R session:

## R input

```

173 install.packages("simmer")

```

Once this package is installed it is loaded using

## R input

```

174 library(simmer)

```

## 2.5 A NOTE ON HOW CODE IS DISPLAYED IN THIS BOOK

### FURTHER READING

- Becskei, A. and Serrano, L. (2000). Engineering stability in gene networks by autoregulation. *Nature*, 405: 590–593.
- Rosenfeld, N., Elowitz, M.B., and Alon, U. (2002). Negative auto-regulation speeds the response time of transcription networks. *J. Mol. Biol.*, 323: 785–793.
- Savageau, M.A. (1976). *Biochemical Systems Analysis: A study of Function and Design in Molecular Biology*. Addison-Wesley. Chap. 16.
- Savageau, M.A. (1974). Comparison of classical and auto-genous systems of regulation in inducible operons. *Nature*, 252: 546–549.





# II

---

## Probabilistic Modelling



# Markov Chains

---

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

## 3.1 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 3.1.

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

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

## 3.2 THEORY

---

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

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

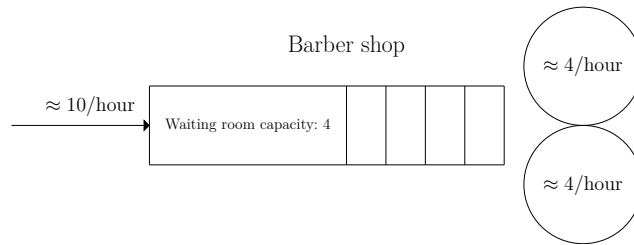


Figure 3.1 Diagrammatic representation of the barber shop as a queuing system.

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

$$S = \{0, 1, 2, 3, 4, 5, 6\} \quad (3.1)$$

As customers arrive and leave the system goes between states as shown in Figure 3.2.

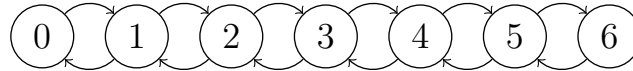


Figure 3.2 Diagrammatic representation of the state space

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

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

For our barber shop we will consider it as a continuous Markov chain as shown in Figure 3.3

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

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

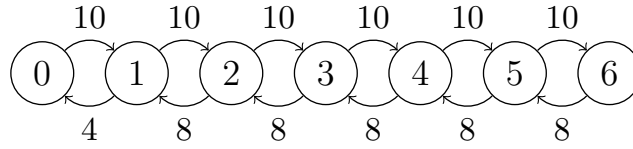


Figure 3.3 Diagrammatic representation of the state space and the transition rates

$$Q = \begin{pmatrix} -10 & 10 & 0 & 0 & 0 & 0 & 0 \\ 4 & -14 & 10 & 0 & 0 & 0 & 0 \\ 0 & 8 & -18 & 10 & 0 & 0 & 0 \\ 0 & 0 & 8 & -18 & 10 & 0 & 0 \\ 0 & 0 & 0 & 8 & -18 & 10 & 0 \\ 0 & 0 & 0 & 0 & 8 & -18 & 10 \\ 0 & 0 & 0 & 0 & 0 & 8 & -8 \end{pmatrix} \quad (3.2)$$

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

We can use  $Q$  to understand the probability of being in a given state after  $t$  time units. This can be represented mathematically using a matrix  $P_t$  where  $(P_t)_{ij}$  is the probability of being in state  $j$  after  $t$  time units having started in state  $i$ . We can use  $Q$  to calculate  $P_t$  using the matrix exponential:

$$P_t = e^{Qt} \quad (3.3)$$

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

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

$$\pi Q = 0 \quad (3.4)$$

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

### 3.3 SOLVING WITH PYTHON

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

## Python input

```

175 def get_transition_rate(
176     in_state, out_state, waiting_room=4, num_barbers=2,
177 ):
178     """Return the transition rate for two given states.
179
180     Args:
181         in_state: an integer
182         out_state: an integer
183         waiting_room: an integer (default: 4)
184         num_barbers: an integer (default: 2)
185
186     Returns:
187         A real.
188     """
189     arrival_rate = 10
190     service_rate = 4
191
192     capacity = waiting_room + num_barbers
193     delta = out_state - in_state
194
195     if delta == 1 and in_state < capacity:
196         return arrival_rate
197
198     if delta == -1:
199         return min(in_state, num_barbers) * service_rate
200
201     return 0

```

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

## Python input

```

202 import itertools
203 import numpy as np
204
205
206 def get_transition_rate_matrix(waiting_room=4, num_barbers=2):
207     """Return the transition matrix  $Q$ .
208
209     Args:
210         waiting_room: an integer (default: 4)
211         num_barbers: an integer (default: 2)
212
213     Returns:
214         A matrix.
215     """
216     capacity = waiting_room + num_barbers
217     state_pairs = itertools.product(
218         range(capacity + 1), repeat=2
219     )
220
221     flat_transition_rates = [
222         get_transition_rate(
223             in_state=in_state,
224             out_state=out_state,
225             waiting_room=waiting_room,
226             num_barbers=num_barbers,
227         )
228         for in_state, out_state in state_pairs
229     ]
230     transition_rates = np.reshape(
231         flat_transition_rates, (capacity + 1, capacity + 1)
232     )
233     np.fill_diagonal(
234         transition_rates, -transition_rates.sum(axis=1)
235     )
236
237     return transition_rates

```

Using this we can obtain the matrix  $Q$  for our default system:

## Python input

```

238 Q = get_transition_rate_matrix()
239 print(Q)

```

which gives:

## Python output

```

240 [[-10  10   0   0   0   0   0]
241 [  4 -14  10   0   0   0   0]
242 [  0   8 -18  10   0   0   0]
243 [  0   0   8 -18  10   0   0]
244 [  0   0   0   8 -18  10   0]
245 [  0   0   0   0   8 -18  10]
246 [  0   0   0   0   0   8 -8]]

```

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

## Python input

```

247 import scipy.linalg
248
249 print(scipy.linalg.expm(Q * 0.5).round(5))

```

which gives:

## Python output

```

250 [[0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.0815 ]
251 [0.08501 0.18292 0.18666 0.1708  0.14377 0.1189  0.11194]
252 [0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346]
253 [0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142]
254 [0.02667 0.07361 0.10005 0.13422 0.17393 0.2189  0.27262]
255 [0.01567 0.0487  0.07552 0.11775 0.17512 0.24484 0.32239]
256 [0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914]]

```

To see what would happen after 500 time units we obtain:



## Python input

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

which gives:

## Python output

```
258 [[0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
259 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
260 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
261 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
262 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
263 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]
264 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]]
```

We see that no matter what state (row) the system is in, after 500 time units the probabilities are all the same. We could in fact stop our analysis here, however our choice of 500 time units was arbitrary and might not be the correct amount for all possible scenarios, as such we will continue to aim to solve the underlying equation 3.4 directly.

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

## Python input

```

265 def get_steady_state_vector(Q):
266     """Return the steady state vector of any given continuous
267     time transition rate matrix.
268
269     Args:
270         Q: a transition rate matrix
271
272     Returns:
273         A vector
274     """
275     state_space_size, _ = Q.shape
276     A = np.vstack((Q.T, np.ones(state_space_size)))
277     b = np.append(np.zeros(state_space_size), 1)
278     x, _, _, _ = np.linalg.lstsq(A, b, rcond=None)
279     return x

```

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

## Python input

```

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

```

we get:

## Python output

```

281 [0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176]

```

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

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

## Python input

```

282 def get_probability_of_full_shop(
283     waiting_room=4, num_barbers=2
284 ):
285     """Return the probability of the barber shop being full.
286
287     Args:
288         waiting_room: an integer (default: 4)
289         num_barbers: an integer (default: 2)
290
291     Returns:
292         A real.
293     """
294     Q = get_transition_rate_matrix(
295         waiting_room=waiting_room, num_barbers=num_barbers,
296     )
297     pi = get_steady_state_vector(Q)
298     return pi[-1]

```

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

## Python input

```

299 print(round(get_probability_of_full_shop(), 6))

```

which gives:

## Python output

```

300 0.261756

```

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

## Python input

```

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

```

which gives:

Python output

```
0.23557
```

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

Python input

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

Python output

```
0.078636
```

### 3.4 SOLVING WITH R

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

## R input

```

305  #' Return the transition rate for two given states.
306  #'
307  #' @param in_state an integer
308  #' @param out_state an integer
309  #' @param waiting_room an integer (default: 4)
310  #' @param num_barbers an integer (default: 2)
311  #'
312  #' @return A real
313  get_transition_rate <- function(in_state,
314                                out_state,
315                                waiting_room = 4,
316                                num_barbers = 2){
317
318    arrival_rate <- 10
319    service_rate <- 4
320
321    capacity <- waiting_room + num_barbers
322    delta <- out_state - in_state
323
324    if (delta == 1) {
325      if (in_state < capacity) {
326        return(arrival_rate)
327      }
328
329      if (delta == -1) {
330        return(min(in_state, num_barbers) * service_rate)
331      }
332      return(0)
333    }

```

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

## R input

```

334  vectorized_get_transition_rate <- Vectorize(
335    get_transition_rate,
336    vectorize.args = c("in_state", "out_state")
337  )

```

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

```

338 #' Return the transition rate matrix Q
339 #'
340 #' @param waiting_room an integer (default: 4)
341 #' @param num_barbers an integer (default: 2)
342 #'
343 #' @return A matrix
344 get_transition_rate_matrix <- function(waiting_room = 4,
345                                     num_barbers = 2){
346   max_state <- waiting_room + num_barbers
347
348   Q <- outer(0:max_state,
349             0:max_state,
350             vectorized_get_transition_rate,
351             waiting_room = waiting_room,
352             num_barbers = num_barbers
353           )
354   row_sums <- rowSums(Q)
355
356   diag(Q) <- -row_sums
357   Q
358 }

```

Using this we can obtain the matrix  $Q$  for our default system:

R input

```

359 Q <- get_transition_rate_matrix()
360 print(Q)

```

which gives:

## R output

```

361      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
362 [1,]  -10  10   0   0   0   0   0
363 [2,]   4 -14  10   0   0   0   0
364 [3,]   0  8 -18  10   0   0   0
365 [4,]   0  0  8 -18  10   0   0
366 [5,]   0  0  0  8 -18  10   0
367 [6,]   0  0  0  0  8 -18  10
368 [7,]   0  0  0  0  0  8 -8

```

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

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

## R input

```

369 library(expm, warn.conflicts = FALSE, quietly = TRUE)
370 library(magrittr, warn.conflicts = FALSE, quietly = TRUE)
371
372 print( (Q * .5) %>% expm %>% round(5))

```

which gives:

## R output

```

373      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
374 [1,] 0.10492 0.21254 0.20377 0.17142 0.13021 0.09564 0.08150
375 [2,] 0.08501 0.18292 0.18666 0.17080 0.14377 0.11890 0.11194
376 [3,] 0.06521 0.14933 0.16338 0.16478 0.15633 0.14751 0.15346
377 [4,] 0.04388 0.10931 0.13183 0.15181 0.16777 0.18398 0.21142
378 [5,] 0.02667 0.07361 0.10005 0.13422 0.17393 0.21890 0.27262
379 [6,] 0.01567 0.04870 0.07552 0.11775 0.17512 0.24484 0.32239
380 [7,] 0.01068 0.03668 0.06286 0.10824 0.17448 0.25791 0.34914

```

After 500 time units we obtain:

## R input

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

which gives:

## R output

```
382      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
383 [1,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
384 [2,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
385 [3,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
386 [4,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
387 [5,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
388 [6,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
389 [7,] 0.03431 0.08577 0.10722 0.13402 0.16752 0.2094 0.26176
```

We see that no matter what state (row) we are in, after 500 time units the probabilities are all the same. We could in fact stop our analysis here, however our choice of 500 time units was arbitrary and might not be the correct amount for all possible scenarios, as such we will continue to aim to solve the underlying equation 3.4 directly.

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



## R input

```

390 library(pracma, warn.conflicts = FALSE, quietly = TRUE)
391
392 #' Return the steady state vector of any given continuous time
393 #' transition rate matrix
394 #'
395 #' @param Q a transition rate matrix
396 #'
397 #' @return A vector
398 get_steady_state_vector <- function(Q){
399   state_space_size <- dim(Q)[1]
400   A <- rbind(t(Q), 1)
401   b <- c(integer(state_space_size), 1)
402   mldivide(A, b)
403 }

```

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

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

## R input

```

404 print(get_steady_state_vector(Q))

```

we get:

## R output

```

405           [,1]
406 [1,] 0.03430888
407 [2,] 0.08577220
408 [3,] 0.10721525
409 [4,] 0.13401906
410 [5,] 0.16752383
411 [6,] 0.20940479
412 [7,] 0.26175598

```

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

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

R input

```

413  #' Return the probability of the barber shop being full
414  #'
415  #' @param waiting_room (default: 4)
416  #' @param num_barbers (default: 2)
417  #'
418  #' @return A real
419  get_probability_of_full_shop <- function(waiting_room = 4,
420                                         num_barbers = 2){
421      arrival_rate <- 10
422      service_rate <- 4
423      pi <- get_transition_rate_matrix(
424          waiting_room = waiting_room,
425          num_barbers = num_barbers
426      ) %>%
427          get_steady_state_vector()
428
429      capacity <- waiting_room + num_barbers
430      pi[capacity + 1]
431  }

```

Now we can run this code efficiently with both scenarios:

R input

```

432  print(get_probability_of_full_shop(waiting_room = 6))

```

which decreases the probability of a full shop to:

R output

```

433  [1] 0.2355699

```

but adding another barber and chair:

R input

434

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

gives:

R output

435

```
[1] 0.0786359
```

### 3.5 RESEARCH

---

TBA



# Discrete Event Simulation

COMPLEX situations further compounded by randomness appear throughout our daily lives. For example, 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, the particular type we are going to cover here is called Discrete Event Simulation.

## 4.1 PROBLEM

Consider the following situation: a bicycle repair shop would like reconfigure their set-up in order to guarantee that all bicycles processed by the repair shop 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, manned by one member of staff who can inspect one bicycle at a time. On average an inspection takes around 3 minutes.
- After inspection it is found that around 20% of bicycles do not need repair, and they are then ready for collection.
- After inspection it is found that around 80% of bicycles go on to be repaired. These then wait in line outside the repair workshop, which is manned 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 4.1

We can also assume that there is infinite capacity at the bicycle repair shop for waiting bicycles. 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?

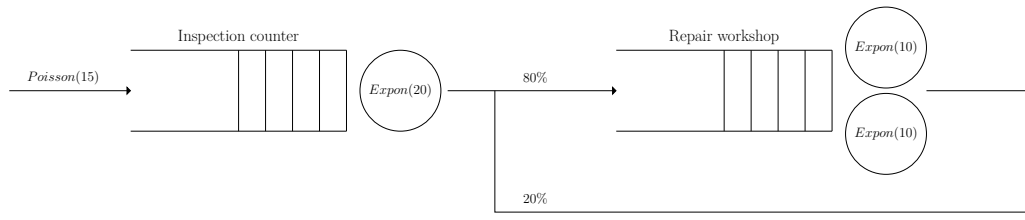


Figure 4.1 Diagrammatic representation of the bicycle repair shop as a queuing system.

## 4.2 THEORY

A number of the events that govern the behaviour 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 arranged in a complex system such as the bicycle shop, using analytical methods to manipulate these probabilities can become difficult. One method to deal with this is *simulation*.

Consider one probabilistic event, rolling a die. A die has six sides numbered 1 to 6, 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 we roll the die a large number of times, we would expect  $\frac{1}{6}$  of those rolls to be a 1. This is called the *law of large numbers*.

Now imagine we have an event in which we do not know the analytical probability of it occurring. 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 we estimate the probability of obtaining a 5 on this die?

Rolling the weighted die once does not give us much information. However due to the law of large numbers, we can roll this die a number of times, and find the proportion of those rolls which gave a 5. The more times we roll the die, the closer this proportion approaches the underlying probability of obtaining a 5.

For a complex system such as the bicycle shop, we would like 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 work this out analytically. So, just like the weighted die, we would like to observe this system a number of times and record the overall proportions of bicycles spending longer than 30 minutes in the shop, which will converge to the true underlying proportion. However 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 this would be costly in terms of money also. We can however build a virtual representation of this complex system on a computer, and observe a virtual day of work much more quickly and much less costly on the computer, 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 to random numbers, that need to be generated.

Computers are deterministic, therefore true randomness is not always possible. 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 we can again manipulate the law of large numbers. Let  $X \sim U(0, 1)$ , a uniformly pseudorandom variable between 0 and 1. Let  $D$  be the outcome of a roll of an unbiased die. Then  $D$  can be defined as:

$$D = \begin{cases} 1 & \text{if } 0 \leq X < \frac{1}{6} \\ 2 & \text{if } \frac{1}{6} \leq X < \frac{2}{6} \\ 3 & \text{if } \frac{2}{6} \leq X < \frac{3}{6} \\ 4 & \text{if } \frac{3}{6} \leq X < \frac{4}{6} \\ 5 & \text{if } \frac{4}{6} \leq X < \frac{5}{6} \\ 6 & \text{if } \frac{5}{6} \leq X < 1 \end{cases} \quad (4.1)$$

The bicycle repair shop is a system made up of interactions of a number of other simpler 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 the the repair workshop,
- the time each those bicycles spends at the repair shop.

As the simulation progresses these events will be generated, and will interact together as described in Section 5.1. 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 just like the weighted die, running this simulation once does not give us much information. But we can run the simulation many times and take an average proportion, to smooth out any variability.

The process outlined above is a particular implementation of Monte Carlo simulation called *discrete event simulation*, which generates pseudorandom numbers and observes their interactions. In practice there are two main approaches to simulating complex probabilistic systems such as this one: the *event scheduling* approach, and *process based* simulation. It just so happens that the main implementations in Python and R use each of these approaches, so you will see both approaches used here.

#### 4.2.1 Event Scheduling Approach

When using the event scheduling approach, we can think of the ‘virtual representation’ of the system as being the facilities that the bicycles use, shown in Figure 4.1.

Then we let 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 events occur that cause further events to occur in the future, either immediately or after a delay, such as after some time in service. 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.

#### 4.2.2 Process Based Simulation

When using process based simulation, we can think of the ‘virtual representation’ of the system as being 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 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.

### 4.3 SOLVING WITH PYTHON

---

In this book we will use the Ciw library in order to conduct discrete event simulation in Python. Ciw uses the event scheduling approach, which means we must define the system’s facilities, and then let customers loose to interact with them.

In this case there are two facilities to define: the inspection desk and the repair workshop. Let’s order these as so. For each of these we need to define:

- 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 we will assume that the time between consecutive arrivals follow a exponential distribution, and that the service times also follow exponential distributions. These are common assumptions for this sort of queueing system.

In Ciw, these are defined in a Network object, created using the `ciw.create_network`



function. The function below creates a Network object that defines the for a given set of parameters bicycle repair shop:

#### Python input

```

436 import ciw
437
438
439 def build_network_object(
440     num_inspectors=1, num_repairers=2,
441 ):
442     """Returns a Network object that defines the repair shop.
443
444     Args:
445         num_inspectors: a positive integer (default: 1)
446         num_repairers: a positive integer (default: 2)
447
448     Returns:
449         a Ciw network object
450     """
451     arrival_rate = 15
452     inspection_rate = 20
453     repair_rate = 10
454     prob_need_repair = 0.8
455     N = ciw.create_network(
456         arrival_distributions=[
457             ciw.dists.Exponential(arrival_rate),
458             ciw.dists.NoArrivals(),
459         ],
460         service_distributions=[
461             ciw.dists.Exponential(inspection_rate),
462             ciw.dists.Exponential(repair_rate),
463         ],
464         number_of_servers=[num_inspectors, num_repairers],
465         routing=[[0.0, prob_need_repair], [0.0, 0.0]],
466     )
467     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

```

468 N = build_network_object()
469 print(N.number_of_nodes)

```

which gives:

## Python output

```

470 2

```

Now we have defined the system, we need to use this to build the virtual representation of the system: in Ciw this is a Simulation object. Once this is built the simulation can be run, that is observe it for one virtual day. The following function does this:

## Python input

```

471 def run_simulation(network, seed=0):
472     """Builds a simulation object and runs it for 8 time units.
473
474     Args:
475         network: a Ciw network object
476         seed: a float (default: 0)
477
478     Returns:
479         a Ciw simulation object after a run of the simulation
480     """
481     max_time = 8
482     ciw.seed(seed)
483     Q = ciw.Simulation(network)
484     Q.simulate_until_max_time(max_time)
485     return Q

```

Notice here a random seed is set. This is because there is some element of randomness when initialising this object, and much randomness in running the simulation, and in order to ensure reproducible results we force the pseudorandom number generator to produce the same sequence of pseudorandom numbers each time. 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.

Now we wish 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. In order to do so, we'll use the `pandas` library for efficient manipulation of data frames.

#### Python input

```

486 import pandas as pd
487
488
489 def get_proportion(Q):
490     """Returns the proportion of bicycles spending over a given
491     limit at the repair shop.
492
493     Args:
494         Q: a Ciw simulation object after a run of the
495         simulation
496
497     Returns:
498         a real
499     """
500     limit = 0.5
501     inds = Q.nodes[-1].all_individuals
502     recs = pd.DataFrame(
503         dr for ind in inds for dr in ind.data_records
504     )
505     recs["total_time"] = (
506         recs["exit_date"] - recs["arrival_date"]
507     )
508     total_times = recs.groupby("id_number")["total_time"].sum()
509     return (total_times > limit).mean()

```

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

Python input

```
510 N = build_network_object()
511 Q = run_simulation(N)
512 p = get_proportion(Q)
513 print(round(p, 6))
```

This piece of code gives

Python output

```
514 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, and an average proportion taken. In order to do so, let's write a function that performs the above experiment over a number of trials, then finds an average proportion:

## Python input

```

515 def get_average_proportion(num_inspectors=1, num_repairers=2):
516     """Returns the average proportion of bicycles spending over
517     a given limit at the repair shop.
518
519     Args:
520         num_inspectors: a positive integer (default: 1)
521         num_repairers: a positive integer (default: 2)
522
523     Returns:
524         a real
525     """
526     num_trials = 100
527     N = build_network_object(
528         num_inspectors=num_inspectors,
529         num_repairers=num_repairers,
530     )
531     proportions = []
532     for trial in range(num_trials):
533         Q = run_simulation(N, seed=trial)
534         proportion = get_proportion(Q=Q)
535         proportions.append(proportion)
536     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

```

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

```

which gives:

## Python output

```

539 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 we wish to compare: 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? Let's investigate. First look at the situation where the additional member of staff works at the inspection desk:

#### Python input

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

which gives:

#### Python output

```
542 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

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

which gives:

#### Python output

```
545 0.103591
```

that is 10.36% of bicycles.

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

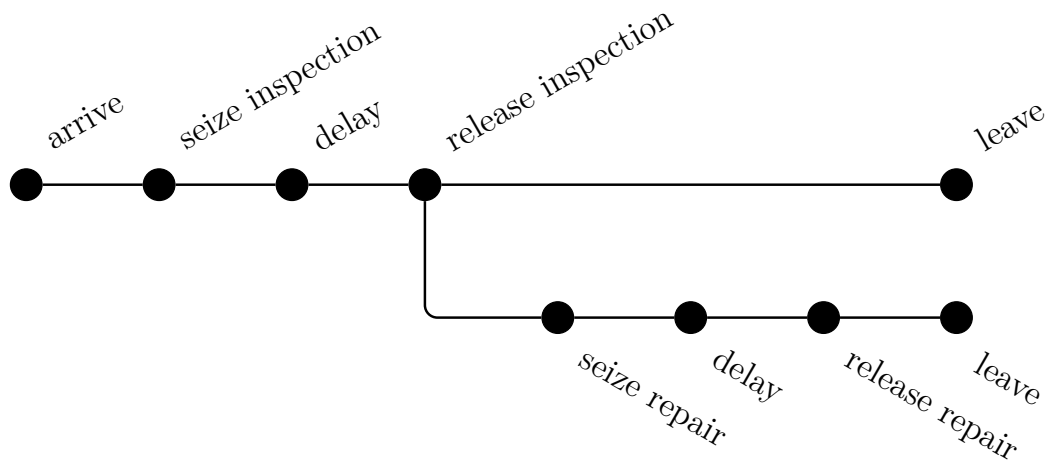


Figure 4.2 Diagrammatic representation of the forked trajectories a bicycle can take

#### 4.4 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 we must define the each bicycle's sequence of actions, 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 ?? shows the branched trajectories than a bicycle would take at the repair shop:

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

## R input

```

546 library(simmer)
547
548 #' Returns a simmer trajectory object outlining the bicycles
549 #' path through the repair shop
550 #'
551 #' @return A simmer trajectory object
552 define_bicycle_trajectories <- function() {
553   inspection_rate <- 20
554   repair_rate <- 10
555   prob_need_repair <- 0.8
556   bicycle <-
557     trajectory("Inspection") %>%
558     seize("Inspector") %>%
559     timeout(function() {
560       rexp(1, inspection_rate)
561     }) %>%
562     release("Inspector") %>%
563     branch(
564       function() (runif(1) < prob_need_repair),
565       continue = c(F),
566       trajectory("Repair") %>%
567         seize("Repairer") %>%
568         timeout(function() {
569           rexp(1, repair_rate)
570         }) %>%
571         release("Repairer"),
572       trajectory("Out")
573     )
574   return(bicycle)
575 }

```

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

```

576 #' Runs one trial of the simulation.
577 #'
578 #' @param bicycle a simmer trajectory object
579 #' @param num_inspectors positive integer (default: 1)
580 #' @param num_repairers positive integer (default: 2)
581 #' @param seed a float (default: 0)
582 #'
583 #' @return A simmer simulation object after one run of
584 #;         the simulation
585 run_simulation <- function(bicycle,
586                           num_inspectors = 1,
587                           num_repairers = 2,
588                           seed = 0) {
589   arrival_rate <- 15
590   max_time <- 8
591   repair_shop <-
592     simmer("Repair Shop") %>%
593     add_resource("Inspector", num_inspectors) %>%
594     add_resource("Repairer", num_repairers) %>%
595     add_generator("Bicycle", bicycle, function() {
596       rexp(1, arrival_rate)
597     })
598
599   set.seed(seed)
600   repair_shop %>% run(until = 8)
601   return(repair_shop)
602 }

```

Notice here a random seed is set. This is because there are elements of randomness when running the simulation, and in order to ensure reproducible results we force the pseudorandom number generator to produce the same sequence of pseudorandom numbers each time. 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.

Now we wish 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. Using Simmer's `get_mon_arrivals()` function we can get a data frame of records to manipulate.

## R input

```

603  #' Returns the proportion of bicycles spending over 30
604  #' minutes in the repair shop
605  #'
606  #' @param repair_shop a simmer simulation object
607  #'
608  #' @return a float between 0 and 1
609  get_proportion <- function(repair_shop) {
610    limit <- 0.5
611    recs <- repair_shop %>% get_mon_arrivals()
612    total_times <- recs$end_time - recs$start_time
613    return(mean(total_times > 0.5))
614  }

```

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

## R input

```

615  bicycle <- define_bicycle_trajectories()
616  repair_shop <- run_simulation(bicycle = bicycle)
617  print(get_proportion(repair_shop = repair_shop))

```

This piece of code gives

## R output

```

618  [1] 0.04032258

```

meaning 4.03% 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, and an average proportion taken. In order to do so, let's write a function that performs the above experiment over a number of trials, then finds an average proportion:

R input

```

619 #' Returns the average proportion of bicycles spending over
620 #' a given limit at the repair shop.
621 #'
622 #' @param num_inspectors positive integer (default: 1)
623 #' @param num_repairers positive integer (default: 2)
624
625 #' @return a float between 0 and 1
626 get_average_proportion <- function(num_inspectors = 1,
627                                   num_repairers = 2) {
628   num_trials <- 100
629   bicycle <- define_bicycle_trajectories()
630   proportions <- c()
631   for (trial in 1:num_trials) {
632     repair_shop <- run_simulation(
633       bicycle = bicycle,
634       num_inspectors = num_inspectors,
635       num_repairers = num_repairers,
636       seed = trial
637     )
638     proportion <- get_proportion(
639       repair_shop = repair_shop
640     )
641     proportions[trial] <- proportion
642   }
643   return(mean(proportions))
644 }

```

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

R input

```

645 print(
646   get_average_proportion(
647     num_inspectors = 1,
648     num_repairers = 2)
649 )

```

which gives:

## R output

```
650 [1] 0.1551579
```

that is, on average 15.52% of bicycles will spend longer than 30 minutes at the repair shop.

Now consider the two possible future scenarios we wish to compare: 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? Let's investigate. First look at the situation where the additional member of staff works at the inspection desk:

## R input

```
651 print(  
652   get_average_proportion(  
653     num_inspectors = 2,  
654     num_repairers = 2)  
655   )
```

which gives:

## R output

```
656 [1] 0.04115338
```

that is 4.12% of bicycles.

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

## R input

```
657 print(  
658   get_average_proportion(  
659     num_inspectors = 1,  
660     num_repairers = 3)  
661   )
```

which gives:

## R output

662

```
[1] 0.1000899
```

that is 10.01% 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.

## 4.5 RESEARCH

---

TBA



# III

---

## Dynamical Systems





# Systems Dynamics

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

## 5.1 PROBLEM

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

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

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

## 5.2 THEORY

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

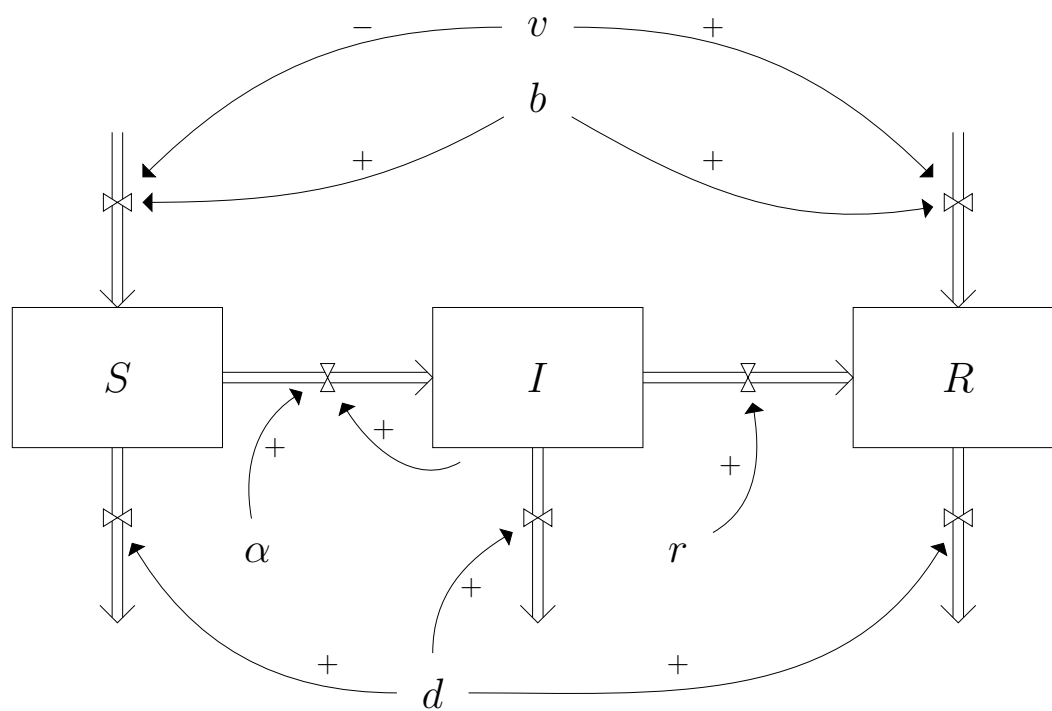


Figure 5.1 Diagrammatic representation of the epidemiology model

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

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

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

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

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

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

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

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

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

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

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

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

### 5.3 SOLVING WITH PYTHON

---

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

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

## Python input

```

663 def derivatives(y, t, vaccine_rate, birth_rate=0.01):
664     """Defines the system of differential equations that
665     describe the epidemiology model.
666
667     Args:
668         y: a tuple of three integers
669         t: a positive float
670         vaccine_rate: a positive float <= 1
671         birth_rate: a positive float <= 1
672
673     Returns:
674         A tuple containing dS, dI, and dR
675     """
676     infection_rate = 0.3
677     recovery_rate = 0.02
678     death_rate = 0.01
679     S, I, R = y
680     N = S + I + R
681     dSdt = (
682         -((infection_rate * S * I) / N)
683         + ((1 - vaccine_rate) * birth_rate * N)
684         - (death_rate * S)
685     )
686     dIdt = (
687         ((infection_rate * S * I) / N)
688         - (recovery_rate * I)
689         - (death_rate * I)
690     )
691     dRdt = (
692         (recovery_rate * I)
693         - (death_rate * R)
694         + (vaccine_rate * birth_rate * N)
695     )
696     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

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

## Python output

```
698 (-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

```

699 from scipy.integrate import odeint
700
701
702 def integrate_ode(
703     derivative_function,
704     t,
705     y0=(2999, 1, 0),
706     vaccine_rate=0.85,
707     birth_rate=0.01,
708 ):
709     """Numerically solve the system of differential equations.
710
711     Args:
712         derivative_function: a function returning a tuple
713             of three floats
714         t: an array of increasing positive floats
715         y0: a tuple of three integers (default: (2999, 1, 0))
716         vaccine_rate: a positive float <= 1 (default: 0.85)
717         birth_rate: a positive float <= 1 (default: 0.01)
718
719     Returns:
720         A tuple of three arrays
721     """
722     results = odeint(
723         derivative_function,
724         y0,
725         t,
726         args=(vaccine_rate, birth_rate),
727     )
728     S, I, R = results.T
729     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%:

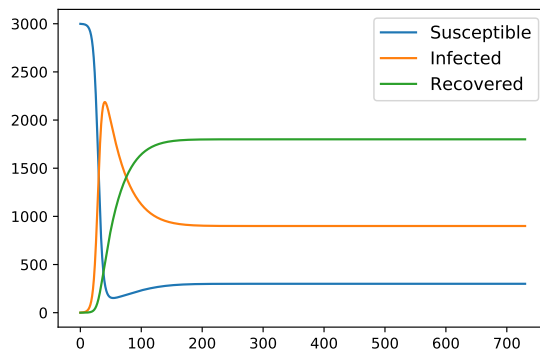


Figure 5.2 Output of code line 724-729

## Python input

```

730 import numpy as np
731 from scipy.integrate import odeint
732
733 t = np.arange(0, 730.01, 0.01)
734 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

```

735 import matplotlib.pyplot as plt
736
737 fig, ax = plt.subplots(1)
738 ax.plot(t, S, label='Susceptible')
739 ax.plot(t, I, label='Infected')
740 ax.plot(t, R, label='Recovered')
741 ax.legend(fontsize=12)
742 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



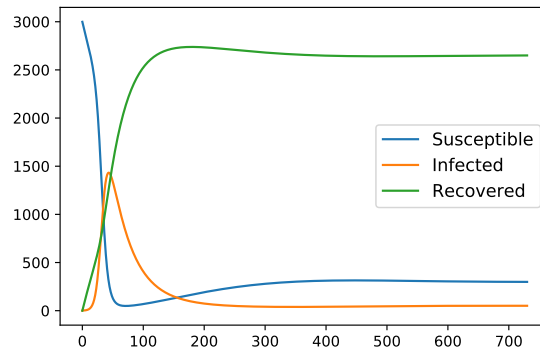


Figure 5.3 Output of code line 732-737

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

```
743 t = np.arange(0, 730.01, 0.01)
744 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

```
745 fig, ax = plt.subplots(1)
746 ax.plot(t, S, label='Susceptible')
747 ax.plot(t, I, label='Infected')
748 ax.plot(t, R, label='Recovered')
749 ax.legend(fontsize=12)
750 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

```

751 def daily_cost(
752     derivative_function=derivatives, vaccine_rate=0.85
753 ):
754     """Calculates the daily cost to the public health system
755     after 2 years.
756
757     Args:
758         derivative_function: a function returning a tuple
759             of three floats
760         vaccine_rate: a positive float <= 1 (default: 0.85)
761
762     Returns:
763         the daily cost
764     """
765     max_time = 730
766     time_step = 0.01
767     birth_rate = 0.01
768     vaccine_cost = 220
769     medication_cost = 10
770     t = np.arange(0, max_time + time_step, time_step)
771     S, I, R = integrate_ode(
772         derivatives,
773         t,
774         vaccine_rate=vaccine_rate,
775         birth_rate=birth_rate,
776     )
777     N = S[-1] + I[-1] + R[-1]
778     daily_vaccine_cost = (
779         N * birth_rate * vaccine_rate * vaccine_cost
780     ) / time_step
781     daily_meds_cost = (I[-1] * medication_cost) / time_step
782     return daily_vaccine_cost + daily_meds_cost

```

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

## Python input

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

which gives

## Python output

```
785 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

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

which gives

## Python output

```
788 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

```

789  #' Defines the system of differential equations that describe
790  #' the epidemiology model.
791  #'
792  #' @param t a positive float
793  #' @param y a tuple of three integers
794  #' @param vaccine_rate a positive float <= 1
795  #' @param birth_rate a positive float <= 1
796  #'
797  #' @return a list containing dS, dI, and dR
798  derivatives <- function(t, y, parameters){
799      infection_rate = 0.3
800      recovery_rate = 0.02
801      death_rate = 0.01
802      with(as.list(c(y, parameters)), {
803          N = S + I + R
804          dSdt = (-((infection_rate * S * I) / N)
805                  + ((1 - vaccine_rate) * birth_rate * N)
806                  - (death_rate * S))
807          dIdt = (((infection_rate * S * I) / N)
808                  - (recovery_rate * I)
809                  - (death_rate * I))
810          dRdt = ((recovery_rate * I)
811                  - (death_rate * R)
812                  + (vaccine_rate * birth_rate * N))
813          list(c(dSdt, dIdt, dRdt))
814      })
815  }

```

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

```

816  derivatives(t=0,
817             y=c(S=4, I=1, R=0),
818             parameters=c(vaccine_rate=0.5, birth_rate=0.01)
819  )

```

## R output

```

820 [[1]]
821 [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

```

822 library(deSolve)
823
824 #' Numerically solve the system of differential equations
825 #'
826 #' @param t an array of increasing positive floats
827 #' @param y0 list of integers (default: c(S=2999, I=1, R=0))
828 #' @param birth_rate a positive float <= 1 (default: 0.01)
829 #' @param vaccine_rate a positive float <= 1 (default: 0.85)
830 #'
831 #' @return a matrix of times, S, I and R values
832 integrate_ode <- function(times,
833                             y0=c(S=2999, I=1, R=0),
834                             birth_rate=0.01,
835                             vaccine_rate=0.84){
836   parameters = c(birth_rate=birth_rate,
837                  vaccine_rate=vaccine_rate)
838   ode(y=y0, times=times, func=derivatives, parms=parameters)
839 }

```

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%:

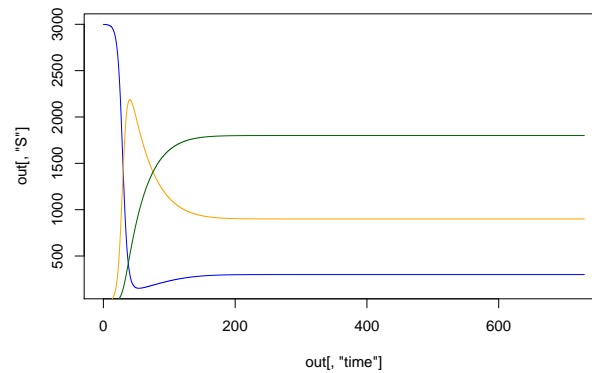


Figure 5.4 Output of code line 832-836

R input

```

840 times <- seq(0, 730, by = 0.01)
841 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.

R input

```

842 pdf("plot_no_vaccine_R.pdf", width=7, height=5)
843 plot(out[, 'time'], out[, 'S'], type='l', col='blue')
844 lines(out[, 'time'], out[, 'I'], type='l', col='orange')
845 lines(out[, 'time'], out[, 'R'], type='l', col='darkgreen')
846 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.

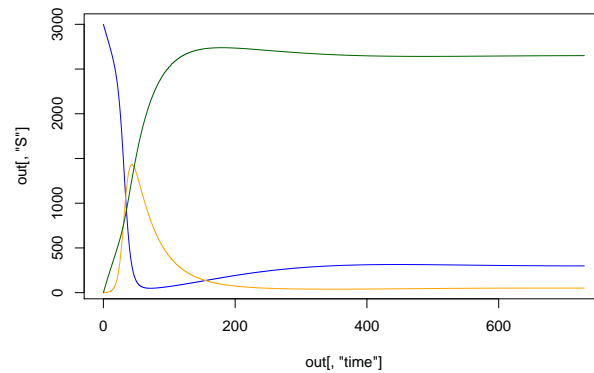


Figure 5.5 Output of code line 839-843

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

```

847 times <- seq(0, 730, by = 0.01)
848 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.

R input

```

849 pdf("plot_with_vaccine_R.pdf", width=7, height=5)
850 plot(out[, 'time'], out[, 'S'], type='l', col='blue')
851 lines(out[, 'time'], out[, 'I'], type='l', col='orange')
852 lines(out[, 'time'], out[, 'R'], type='l', col='darkgreen')
853 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

```

854 #' Calculates the daily cost to the public health
855 #' system after 2 years
856 #'
857 #' @param derivative_function: a function returning a
858 #'                               list of three floats
859 #' @param vaccine_rate: a positive float <= 1 (default: 0.85)
860 #'
861 #' @return the daily cost
862 daily_cost <- function(derivative_function=derivatives,
863                        vaccine_rate=0.85){
864   max_time = 730
865   time_step = 0.01
866   birth_rate = 0.01
867   vaccine_cost = 220
868   medication_cost = 10
869   times <- seq(0, max_time, by = time_step)
870   out <- integrate_ode(times, vaccine_rate=vaccine_rate)
871   N = sum(tail(out[, c('S', 'I', 'R')], n=1))
872   daily_vaccine_cost <- (N
873                        * birth_rate
874                        * vaccine_rate
875                        * vaccine_cost) / time_step
876   daily_medication_cost <- ((tail(out[, 'I'], n=1)
877                          * medication_cost)) / time_step
878   daily_vaccine_cost + daily_medication_cost
879 }

```

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

R input

```

880 cost <- daily_cost(vaccine_rate=0.0)
881 print(cost)

```

which gives

R output

```
882 [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

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

which gives

R output

```
885 [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

---

---

# Bibliography

---

- [1] Hadley Wickham. *Advanced r*. Chapman and Hall/CRC, 2014.

