Half Title Page

Title Page

LOC Page

Vince: to Riggins
Geraint: also, to Riggins

Contents

Foreword

This is the foreword

Preface

This is the preface.

Contributors

Michaél Aftosmis
NASA Ames Research Center
Moffett Field, California

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

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

Gabrielle Allen
Louisiana State University
Baton Rouge, Louisiana

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

Steven F. Ashby
Lawrence Livermore National Laboratory
Livermore, California

David A. Bader
Georgia Institute of Technology
Atlanta, Georgia

Benjamin Bergen
Los Alamos National Laboratory
Los Alamos, New Mexico

Jonathan W. Berry
Sandia National Laboratories
Albuquerque, New Mexico

Martin Berzins
University of Utah
Salt Lake City, Utah

Abhinav Bhatele
University of Illinois
Urbana-Champaign, Illinois

Christian Bischof
RWTH Aachen University
Germany

Rupak Biswas
NASA Ames Research Center
Moffett Field, California

Eric Bohm
University of Illinois
Urbana-Champaign, Illinois

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

Geörge Bosilca
University of Tennessee
Knoxville, Tennessee

Grèg L. Bryan
Columbia University
New York, New York

Marian Bubak
AGH University of Science and Technology
Kraków, Poland

Andrew Canning
Lawrence Berkeley National Laboratory
Berkeley, California

Jonathan Carter
Lawrence Berkeley National Laboratory
Berkeley, California

Zizhong Chen
Jacksonville State University
Jacksonville, Alabama

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

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

Roxana E. Diaconescu
Yahoo! Inc.
Burbank, California

I

Getting Started

CHAPTER 1

Introduction

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

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

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

1.1 WHO IS THIS BOOK FOR?

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

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

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

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

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

1.2 WHAT DO WE MEAN BY APPLIED MATHEMATICS?

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

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

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

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

1.3 WHAT IS OPEN SOURCE SOFTWARE

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

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

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

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

1.4 HOW TO GET THE MOST OUT OF THIS BOOK

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

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

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

Every chapter has the following structure:

1. Introduction - a brief overview of a given problem type. Here we will describe the problem at hand in general terms.

2. An Example problem. This will provide a tangible example problem that offers the reader some intuition for the rest of the discussion.

3. Solving with Python. We will describe the mathematical tools available to us in a programming language called Python to solve the problem.

4. Solving with R. Here we will do the same with the R programming language.

5. Brief theoretic background with pointers to reference texts. Some readers might like to delve in to the mathematics of the problem a bit further, we will include those details here.

6. Examples of research using these methods. Finally, some readers might even be interested in finding out a bit more of what mathematicians are doing on these problems. Often this will include some descriptions of the problem considered but perhaps at a much larger scale than the one presented in the example.

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

CHAPTER 2

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

```
ls
```

On Windows

Cli input

```
dir
```

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

Cli input

```
pwd
```

On Windows

Cli input

```
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

```
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.

- 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

```
>>> 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

```
R
```

This should make something like the following appear:

Cli output

```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

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

R input

```
> 2 + 2
```

When you press Enter, this will give:

R output

```
4
```

To quit R's REPL type the following:

R input

```
> 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

```
> 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

```
python main.py
```

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

Cli input

```
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

```
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 on 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

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

which will give:

Python output

```
8
```

We can also assign values to a variable:

Python input

```python
a = 3
b = 5
c = a + b
print(c)
```

This makes a point at 3 etc...

which will give:

Python output

```
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

```
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

```
a = 3
b = 5
if a < b:
    a = a + 3
elif a > b:
    b = b + 5
else:
    a = a + 3
    b = b + 3
print(a, b)
```

which gives:

Python output

```
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

```python
a = 3
b = 5
while a != b:
    if a < b:
        a = a + 1
    else:
        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

```python
def add_3_to_smallest(a, b):
    """
    This function adds 3 to the smallest of a or b.
    """
    if a < b:
        return a + 3, b
    return a, b + 3
```

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

Python input

```python
print(add_3_to_smallest(a=5, b=-42))
```

which gives:

Python output

```
(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

```python
tennis_players = ["Federer", "S. Williams", "V. Williams", "King"]
```

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

```python
for name in tennis_players:
    print(name)
```

which gives:

Python output

```
Federer
S. Williams
V. Williams
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

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

which will give:

Python output

```
30
33
36
39
42
45
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

```
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

```
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

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

which will give:

R output

```
[1] 8
```

We can also assign values to a variable:

R input

```r
a <- 3
b <- 5
c <- a + b
print(c)
```

which will give:

R output

```
[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

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

giving:

R output

```
[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

```
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

```
a <- 3
b <- 5
if (a < b) {
  a <- a + 3
} else if (a > b) {
  b <- b + 3
} else {
  a <- a + 3
  b <- b + 3
}
print(c(a, b))
```

which gives:

R output

```
[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

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

which gives:

R output

```
[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

```
a <- 3
b <- 5
while (a != b) {
  if (a < b) {
    a <- a + 1
  }
  else {
    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.

R input

```r
add_3_to_smallest <- function(a, b) {
  # This function adds 3 to the smallest of a or b.
  if (a < b) {
    return(c(a + 3, b))
  }
  else {
    return(c(a, b + 3))
  }
}
```

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

```r
add_3_to_smallest <- function(a, b) {
  # This function adds 3 to the smallest of a or b.
  if (a < b) {
    c(a + 3, b)
  }
  else {
    c(a, b + 3)
  }
}
```

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

R input

```r
print(add_3_to_smallest(a = 5, b = -42))
```

which gives:

R output

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

It is possible to iterate over elements inside R vectors:

R input

```
tennis_players <- c("Federer", "S. Williams", "V. Williams", "King")
```

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

R input

```
for (name in tennis_players) {
    print(name)
}
```

which gives:

R output

```
[1] "Federer"
[1] "S. Williams"
[1] "V. Williams"
[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

```
for (i in seq(30, 50, 3)) {
  print(i)
}
```

which will give:

R output

```
[1] 30
[1] 33
[1] 36
[1] 39
[1] 42
[1] 45
[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

```
install.packages("simmer")
```

Once this package is installed it is loaded using

R input

```
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

CHAPTER 3

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

- Barber shop - 2 chairs and 2 barbers - Room for 4 people to wait - Add room for 2 people to wait or add 1 more barber + chair? - What is the probability of having to wait?

3.2 THEORY

- System can be described using a Markov chain which is a collection of States, transitions etc... - In this instance, it is a type of birth-death process that can be drawn... - Has a Matrix representation: Q - We can calculate the expected smallest time for a change to occur. - There is a mathematical process that discretises this system to give us P and the corresponding picture... - In Delta t time what is the probability... - Now we can think about P pi, ..., P to a high power will give us... this will correspond to pi P = pi (this is an eigenvalue problem). - Return to continuous system.

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

```python
def get_transition_rate(
    in_state,
    out_state,
    waiting_room=4,
    number_of_barbers=2,
    arrival_rate=10,
    service_rate=4,
):
    """
    Return the transition rate for two given states.

    For a given barber shop problem this returns the transition rate between two
    states. As well as taking the two states as inputs it takes a number of
    parameters that define the system.

    Args:
        in_state: an integer denoting the current state
        out_state: an integer denoting the next state
        waiting_room: an integer denoting the size of the waiting room (default: 4)
        number_of_barbers: an integer denoting the number of barber and chairs (defaul
        arrival_rate: a real number denoting the number of individuals per unit time
                      that arrive at the barber shop (default: 10)
        service_rate: a real number denoting the number of individuals per unit time
                      that a single barber can serve (default: 4)

    Returns:
        A real.
    """
    capacity = waiting_room + number_of_barbers
    delta = out_state - in_state

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

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

    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, so we need to import it:

Python input

```python
import numpy as np
```

Now we define the function:

Python input

```python
def get_transition_rate_matrix(
    waiting_room=4,
    number_of_barbers=2,
    arrival_rate=10,
    service_rate=4,
):
    """
    Return the transition matrix Q.

    For a given barber shop problem this returns the transition rate matrix
    As inputs it takes a number of
    parameters that define the system.


    Args:
        waiting_room: an integer denoting the size of the waiting room (default: 4)
        number_of_barbers: an integer denoting the number of barber and chairs (defaul
        arrival_rate: a real number denoting the number of individuals per unit time
                      that arrive at the barber shop (default: 10)
        service_rate: a real number denoting the number of individuals per unit time
                      that a single barber can serve (default: 4)

    Returns:
        A matrix.
    """
    capacity = waiting_room + number_of_barbers
    state_pairs = itertools.product(range(capacity + 1), repeat=2)
    flat_transition_rates = [get_transition_rate(
        in_state=in_state,
        out_state=out_state,
        waiting_room=waiting_room,
        number_of_barbers=number_of_barbers,
        arrival_rate=arrival_rate,
        service_rate=service_rate,
    ) for in_state, out_state in state_pairs]
    transition_rates = np.reshape(flat_transition_rates, (capacity + 1, capacity + 1))
    np.fill_diagonal(transition_rates, - transition_rates.sum(axis=1))

    return transition_rates
```

- numpy array - function to generate Q - function to discretise Q - multiple P by

vectors - raise P to high power - write function to solve matrix equation and return pfull - eigenvalue approach
- Run for both scenarios

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

```r
#' Return the transition rate for two given states.
#'
#' @description
#' For a given barber shop problem this returns the transition rate between two
#' states. As well as taking the two states as inputs it takes a number of
#' parameters that define the system.
#'
#' @param in_state an integer denoting the current state
#' @param out_state an integer denoting the next state
#' @param waiting_room an integer denoting the size of the waiting room
#' (default: 4)
#' @param number_of_barbers an integer denoting the number of barber and chairs
#' (default: 2)
#' @param arrival_rate a real number denoting the number of individuals per
#' unit time that arrive at the barber shop (default: 10)
#' @param service_rate a real number denoting the number of individuals per unit
#' time that a single barber can serve (default: 4)
#'
#' @return A real
get_transition_rate <- function(
                                in_state,
                                out_state,
                                waiting_room = 4,
                                number_of_barbers = 2,
                                arrival_rate = 10,
                                service_rate = 4) {
  capacity <- waiting_room + number_of_barbers
  delta <- out_state - in_state

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

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

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

R input

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

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

R input

```r
#' Return the transition rate matrix Q
#'
#' @description
#' For a given barber shop problem this returns the transition rate matrix
#' As inputs it takes a number of
#' parameters that define the system.
#'
#' @param waiting_room an integer denoting the size of the waiting room
#' (default: 4)
#' @param number_of_barbers an integer denoting the number of barber and chairs
#' (default: 2)
#' @param arrival_rate a real number denoting the number of individuals per
#' unit time that arrive at the barber shop (default: 10)
#' @param service_rate a real number denoting the number of individuals per unit
#' time that a single barber can serve (default: 4)
#'
#' @return A matrix
get_transition_rate_matrix <- function(
                                       waiting_room = 4,
                                       number_of_barbers = 2,
                                       arrival_rate = 10,
                                       service_rate = 4) {
  max_state <- waiting_room + number_of_barbers

  Q <- outer(0:max_state,
    0:max_state,
    vectorized_get_transition_rate,
    waiting_room = waiting_room,
    number_of_barbers = number_of_barbers,
    arrival_rate = arrival_rate,
    service_rate = service_rate
  )
  row_sums <- rowSums(Q)

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

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

R input

```
Q <- get_transition_rate_matrix()
Q
```

which gives:

R output

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

One immediate thing 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`:

R input

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

To be able to make use of the nice %>% ”pipe” operator we are also going to load the `dplyr` library:

R input

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

Now if we wanted to see what would happen after 500 time units we obtain:

R input

```
(t(Q) * 500) %>% expm %>% round(6)
```

which gives:

R output

```
          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]
[1,] 0.034309 0.034309 0.034309 0.034309 0.034309 0.034309 0.034309
[2,] 0.085772 0.085772 0.085772 0.085772 0.085772 0.085772 0.085772
[3,] 0.107215 0.107215 0.107215 0.107215 0.107215 0.107215 0.107215
[4,] 0.134019 0.134019 0.134019 0.134019 0.134019 0.134019 0.134019
[5,] 0.167524 0.167524 0.167524 0.167524 0.167524 0.167524 0.167524
[6,] 0.209405 0.209405 0.209405 0.209405 0.209405 0.209405 0.209405
[7,] 0.261756 0.261756 0.261756 0.261756 0.261756 0.261756 0.261756
```

We see that no matter what state (column) 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 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

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

#' Return the steady state vector of any given continuous time transition rate
#' matrix
#'
#' @description
#' Solves the matrix equation Ax=b where A is the matrix Q with an extra row of
#' ones (to ensure a probability vector is given) and b is a vector of 0s and a
#' single one.
#'
#' @param Q a transition rate matrix
#'
#' @return A vector
get_steady_state_vector <- function(Q){
  state_space_size <- dim(Q)[1]
  A <- rbind(t(Q), 1)
  b <- c(integer(state_space_size), 1)
  mldivide(A, b)
}
```

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

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

R input

```r
get_steady_state_vector(Q)
```

we get:

R output

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

```r
#' Return the probability of the barber shop being full
#'
#' @description
#' For a given barber shop problem this returns the probability of it being full
#' As inputs it takes a number of
#' parameters that define the system.
#'
#' @param waiting_room an integer denoting the size of the waiting room
#' (default: 4)
#' @param number_of_barbers an integer denoting the number of barber and chairs
#' (default: 2)
#' @param arrival_rate a real number denoting the number of individuals per
#' unit time that arrive at the barber shop (default: 10)
#' @param service_rate a real number denoting the number of individuals per unit
#' time that a single barber can serve (default: 4)
#'
#' @return A real
get_probability_of_full_shop <- function(
                                         waiting_room = 4,
                                         number_of_barbers = 2,
                                         arrival_rate = 10,
                                         service_rate = 4) {
  pi <- get_transition_rate_matrix(
    waiting_room = waiting_room,
    number_of_barbers = number_of_barbers,
    arrival_rate = arrival_rate,
    service_rate = service_rate
  ) %>%
    get_steady_state_vector()

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

Now we can run this code efficiently with both scenarios:

R input

```r
get_probability_of_full_shop(waiting_room = 6)
```

which decreases the probability of a full shop to:

R output

```
[1] 0.2355699
```

but adding another barber and chair:

R input

```r
get_probability_of_full_shop(number_of_barbers = 3)
```

gives:

R output

```
[1] 0.0786359
```

In fact even with room for 20 people to wait the only way 2 barbers would be able to have a less than 10% of the shop being full is to find a way to each serve .4 more of a customer per hour:

R input

```r
get_probability_of_full_shop(waiting_room = 20, service_rate = 4.5)
```

R output

```
[1] 0.1103209
```

3.5 RESEARCH

TBA

Bibliography

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